arXiv:1509.00239v2 [cs.CR] 4 May 2016

CASH: A Cost Asymmetric Secure Hash Algorithm for Optimal Password Protection

Jeremiah Blocki
Microsoft Research

Anupam Datta
Carnegie Mellon University

May 6, 2016

Abstract

An adversary who has obtained the cryptographic hash of a user's password can mount an offline attack to crack the password by comparing this hash value with the cryptographic hashes of likely password guesses. This offline attacker is limited only by the resources he is willing to invest to crack the password. Key-stretching techniques like hash iteration and memory hard functions have been proposed to mitigate the threat of offline attacks by making each password guess more expensive for the adversary to verify. However, these techniques also increase costs for a legitimate authentication server. We introduce a novel Stackelberg game model which captures the essential elements of this interaction between a defender and an offline attacker. In the game the defender first commits to a key-stretching mechanism, and the offline attacker responds in a manner that optimizes his utility (expected reward minus expected guessing costs). We then introduce Cost Asymmetric Secure Hash (CASH), a randomized key-stretching mechanism that minimizes the fraction of passwords that would be cracked by a rational offline attacker without increasing amortized authentication costs for the legitimate authentication server. CASH is motivated by the observation that the legitimate authentication server will typically run the authentication procedure to verify a correct password, while an offline adversary will typically use incorrect password guesses. By using randomization we can ensure that the amortized cost of running CASH to verify a correct password guess is significantly smaller than the cost of rejecting an incorrect password. Using our Stackelberg game framework we can quantify the quality of the underlying CASH running time distribution in terms of the fraction of passwords that a rational offline adversary would crack. We provide an efficient algorithm to compute high quality CASH distributions for the defender. Finally, we analyze CASH using empirical data from two large scale password frequency datasets. Our analysis shows that CASH can significantly reduce (up to 50%) the fraction of password cracked by a rational offline adversary.

1 Introduction

In recent years the authentication servers at major companies like eBay, Zappos, Sony, LinkedIn and Adobe [3–8] have been breached. These breaches have resulted in the release of the cryptographic hashes of millions of user passwords, each of which has significant economic value to adversaries [36, 60]. An adversary who has obtained the cryptographic hash of a user's password can mount a fully automated attack to crack the user's password by comparing this hash value to the cryptographic hashes of likely password guesses [31]. This offline attacker can try as many password guesses as he likes; he is only limited by the resources that he is willing to invest to crack the password.

Offline attacks are becoming increasingly dangerous due to a combination of several different factors. First, improvements in computing hardware make password cracking cheaper (e.g., [60]). Second, empirical data indicates that many users tend to select low entropy passwords [20, 32, 56]. Finally, offline adversaries now have a wealth of training data available from previous password breaches [37] so the adversary often has very accurate background knowledge about the structure of popular passwords.

Password hash functions like PBKDF2 [43], BCRYPT [54], Argon2 [12] and SCRYPT [51] employ key-stretching [48] to make it more expensive for an offline adversary to crack a hashed password. While key-stretching may reduce the number of password guesses that the adversary is able to try, the legitimate authentication server faces a basic trade-off: he must also pay an increased cost every time a user authenticates.

The basic observation behind our work is that it is possible for the legitimate authentication server to use randomization to gain an advantage in this cat-and-mouse game. The offline adversary will spend most of his time guessing incorrect passwords, while the authentication server will primarily authenticate users with correct passwords. Therefore, it would be desirable to have an authentication procedure whose cost is asymmetric. That is the cost of rejecting an incorrect password is greater than the cost of accepting a correct password. This same basic observation lay behind Manber's proposal to use secret salt values (e.g., "pepper") [46]. For example, the server might store the cryptographic hash $\mathbf{H}(pwd, t)$ for a uniformly random value $t \in \{1, \ldots, m\}$ called the "pepper". An offline adversary will need to compute the hash function m times in total to reject an incorrect password pwd', while the legitimate authentication server will only need to compute it $\frac{m+1}{2}$ times on average to accept a correct password.

We introduce Cost Asymmetric Secure Hash (CASH) a mechanism for protecting passwords against offline attacks while minimizing expected costs to the legitimate authentication server. CASH may be viewed as a simple, yet powerful, extension of [46] in which the distribution over t is not-necessarily uniform — the "peppering" idea of Manber [46] is a special case of our mechanism in which the distribution over t is uniform.

In this paper we seek to address the following questions: How can we quantify the security gains (losses) from the use of secret salt values? What distribution over the secret salt value (t) is optimal for the authentication server? Is there an efficient algorithm to compute this distribution? Does CASH perform better than "pepper" or deterministic key stretching?

Contributions We first introduce a Stackelberg (leader-follower) game which captures the essential aspects of our password setting. Our Stackelberg model can provide helpful guidance for the authentication server by predicting whether or not (a particular level of) key-stretching will significantly reduce the number of passwords that would be cracked by a rational offline adversary in the event of a server breach. In our Stackelberg game the authentication server (leader) first commits to a password hashing strategy, and the offline adversary (follower) gets to play his best response to the server's (leader's) action. That is the adversary selects a threshold B and begins guessing passwords until he either 1) cracks the user's password, or 2) gives up after expending B units of work. The adversary will select a threshold B that maximizes his utility.[1]

Next we give an efficient algorithm for computing good strategies for the leader (authentication server) in this Stackelberg game. The defender wants to find a distribution $\tilde{p}_1 \geq \ldots \geq \tilde{p}_m \geq 0$ over the secret running time parameter $t \in \{1, \ldots, m\}$, which minimizes the number of passwords that an offline adversary would crack. When choosing this distribution, the defender is given a constraint (e.g., $\mathbb{E}[t] = \sum_{t=1}^{m} t \cdot \tilde{p}_t \leq C_{max}$) bounding the server's amortized authentication costs.

Unfortunately, there are no known polynomial time algorithms to compute the Stackelberg equilibrium of our game as this problem reduces to a non-convex optimization problem.[2] However, we develop an efficient algorithm to solve a closely related goal: find the CASH distribution which minimizes the success rate of an adversary with a fixed budget B per user. While this new goal is not equivalent to the Stackelberg equilibrium our experimental results indicate that the resulting CASH distributions translate to good strategies in the original Stackelberg game. At a technical level we show that this new optimization problem can be expressed as a linear program. The key technical challenge in solving this linear program is that it has exponentially many constraints. Fortunately, this linear program can still be solved in polynomial time using an efficient separation oracle that we develop. We also develop a practical algorithm which can quickly

[1] Intuitively, the adversary's utility is his expected reward (the value of a cracked password times the probability he cracks it) minus his expected guessing costs (given by the expected number of times that the adversary needs to evaluate the hash function before he succeeds or gives up).

[2] By contrast, fixing any CASH distribution $\tilde{p}_i \geq \ldots \geq \tilde{p}_m$ it is easy to compute the adversary's best response.

find the (approximately) optimal CASH distribution against a budget B adversary. The algorithm is efficient enough to run on large real world instances (e.g., a dataset of 70 million passwords).

Finally, we evaluated CASH using password frequency data from the RockYou password breach and from a (perturbed) dataset of 70 million Yahoo! passwords [16, 20]. Our analysis shows that CASH significantly outperforms the traditional (deterministic) key-stretching defense as well as the "peppering" defense of [46]. In some instances, CASH reduced the fraction of passwords cracked by a rational adversary by about 50% in comparison to both pepper and traditional key-stretching algorithms.

2 Background

Before we introduce the basic CASH mechanism it is necessary to introduce some notation (Section 2.1) and review the traditional password based authentication process (Section 2.2).

2.1 Notation.

We use $\mathbf{H}$ to denote a cryptographic hash function and we let $\mathbf{Cost}(\mathbf{H})$ denote the cost of evaluating $\mathbf{H}$ one time. To simplify the presentation we will assume that all other costs have been scaled so that $\mathbf{Cost}(\mathbf{H}) = 1$. We use $\mathbf{H}^k$ to denote a hash function that is k-times as expensive to compute.[3] We use $\mathcal{P}$ to denote the space of passwords that users may select, and we use n to denote the number of passwords in this space. We use p_i to denote the probability that a random user selects the password $pwd_i \in \mathcal{P}$. For notational convenience, we assume that the passwords have been sorted so that $p_1 \geq ... \geq p_n$. Given a set S we will write $x \xleftarrow{\$} S$ to denote a uniformly random sample from the set S.

Table 1 contains a summary of the notation used throughout this paper. Some of this notation will be introduced later in the paper when it is first used.

2.2 Traditional Password Authentication.

We begin by giving a brief overview of the traditional password authentication process. Suppose that a user registers for an account with username u and password $pwd_u \in \mathcal{P}$. Typically, an authentication server will store a record like the following $\left(u, s_u, k, \mathbf{H}^k\left(pwd_u, s_u\right)\right)$. Here, $s_u \xleftarrow{\$} \{0,1\}^L$ is a random L-bit salt [9] value used to prevent rainbow table attacks [49] and the parameter k controls the cost of the hash function. We stress that the salt value s_u and the cost parameter k are stored on the server in the clear so an adversary who breaches the authentication server will learn both of these values. We use the notation s_u to emphasize that this salt value is different for each user u. The parameter k is selected subject to the constraint that $k \leq C_{max}$ — the maximum amortized cost that the authentication server is willing to incur for authentication.[4]

When the user authenticates he will type in his username u and a password guess $pwd'_u \in \mathcal{P}$. The authentication server first finds the record $\left(u, s_u, k, \mathbf{H}^k\left(pwd_u, s_u\right)\right)$. It then computes $\mathbf{H}^k\left(pwd'_u, s_u\right)$ and verifies that it matches the stored hash value $\mathbf{H}^k\left(pwd_u, s_u\right)$. Note that authentication will always be successful when the user's password is correct (e.g., $pwd'_u = pwd_u$) because the hash values $\mathbf{H}^k\left(pwd'_u, s_u\right)$ and $\mathbf{H}^k\left(pwd_u, s_u\right)$ must match in this case. Similarly, if the user's password is incorrect (e.g., $pwd_u \neq pwd'_u$) then

[3] In this work we will not focus on the lower level issue of which key-stretching techniques are used. However, this is an important research area [1] and we would strongly advocate for the use of modern key-stretching techniques like memory hard functions. BCRYPT [54] and PBKDF2 [43], use hash iteration for key-stretching. In this case the cost parameter k specifies the number of hash iterations. For example, if $k = 2$ the authentication server would store the tuple $\left(k = 2, \mathbf{H}(\mathbf{H}(pwd))\right)$. The disadvantage to this approach is that a hash function $\mathbf{H}$ might cost orders of magnitude less to evaluate on an Application Specific Integrated Circuit than it would cost to evaluate on a more traditional architecture. By contrast, memory costs tend to be relatively stable across different architectures [33], which motivates the use of memory hard functions for password hashing [50]. Argon2 [12], winner of the recently completed password hashing competition [1], and SCRYPT [51] use memory hard functions to perform key-stretching. In this paper we will simply use $\mathbf{H}^k$ is k-times as expensive to compute without worrying about the specific key-stretching techniques that were employed to achieve this property.

[4] In the traditional (deterministic) key-stretching setting it is clear the hash cost parameter $k = C_{max}$ is equivalent to the maximum authentication cost parameter C_{max}. However, this equivalence will not hold one we introduce a randomized running time parameter t. Thus, it is helpful to use separate notation to separate these distinct parameters.

authentication will fail with high probability because the cryptographic hash function $\mathbf{H}$ is collision resistant.

Server Cost. Under this traditional password mechanism the cost of verifying/rejecting a password is simply k. The authentication server can increase guessing costs for an offline adversary by increasing k, but in doing so the authentication server will increase its own authentication costs proportionally.

Authentication Time Increase. By increasing the cost parameter k the authentication server might potentially increase delay times for the user — especially if key-stretching is performed on a sequential computer. Bonneau and Schechter [22] estimated that $\mathbf{Cost}(\mathbf{H}) \approx \$7 \times 10^{-15}$ for the SHA-256 hash function based on observations of the Bitcoin network. A modern CPU can evaluate SHA-256 around 10^7 times per second so an authentication server who uses hash iteration for key-stretching would need to select $k \leq 10^7$ if he wants to ensure that user delay is at most one second. In this case we would seem to have an upperbound $\mathbf{Cost}\left(\mathbf{H}^k\right) \leq \$7 \times 10^{-8}$ on the cost of a hash function that can be evaluated in 1 second. Fortunately, this bound only applies to naive hash iteration[5]. More effective key-stretching techniques could be used to increase $\mathbf{Cost}\left(\mathbf{H}^k\right)$ by several orders of magnitude (e.g., $\mathbf{Cost}\left(\mathbf{H}^k\right) \geq \$10^{-5}$) without imposing longer authentication delays on the user (even if key-stretching is performed on a sequential computer). For example, the SCRYPT [51] and Argon2 [12] hash functions were intentionally designed to use a larger amount of memory so that it is not possible to (significantly) reduce hashing costs by developing customized hardware. Additionally, Argon2 [12], winner of the password hashing competition, has an optional parameter that would allow the authentication server to exploit parallelism to further reduce the amount of time necessary to perform key-stretching.

2.3 Adversary Model

We consider an untargeted offline attacker whose goal is to break as many passwords as possible. An offline attacker has breached the authentication server and has access to all of the data stored on the server. In the traditional authentication setting an offline adversary learns the tuple $\left(u, s_u, k, \mathbf{H}^k\left(pwd_u, s_u\right)\right)$ for each user u. The adversary will also learn the hash function $\mathbf{H}$ since the code to compute $\mathbf{H}$ is present on the authentication server. We assume that the adversary only uses $\mathbf{H}$ in a blackbox manner (e.g., the adversary can query $\mathbf{H}$ as a random oracle, but he cannot invert $\mathbf{H}$). In general we assume the adversary will obtain the source code for any other procedures that are used during the authentication process. While the authentication server can limit the number of guesses that an online adversary can make (e.g., by locking the adversary out after three incorrect guesses), the authentication server cannot directly limit the number of guesses that an offline attacker can try. An offline attacker is limited only by the resources that s/he is willing to invest trying to crack the user's password.

We assume that the adversary has a value v_u for cracking user u's password. An untargeted offline attacker has the same value $v_u = v$ for every user u. Symantec recently reported that passwords sell for between \$4 and \$30 on the black market [36] so we might reasonably estimate that $v \in [\$4, \$30]$.[6]

We also assume that the adversary knows the empirical password distribution $p_1 \geq ... \geq p_n$ over user selected passwords as well as the corresponding passwords $pwd_1, \ldots, pwd_n$. Thus, the adversary knows that a random user will select pwd_1 with probability p_1, but the adversary does not know which users selected pwd_1.

The adversary will select a threshold B and check (up to) B passwords. In this case the fraction of passwords that the offline adversary will break is at most $\sum_{i=1}^{B} p_i$. Equality holds when the offline adversary adopts his optimal guessing strategy and checks the B most likely passwords $pwd_1, \ldots, pwd_B$. In this case

[5] As we previously noted hash iteration alone is not a particularly effective key-stretching technique. The cost of computing SHA-256 can be reduced by a factor of about 1 million on customize hardware — e.g., see `https://bitcoinmagazine.liberty.me/bitmain-announces-launch-of-next-generation-antminer-s7-bitcoin-miner/` (Retrieved 5/4/2016). Furthermore, we note that modern Bitcoin miners already use Application Specific Integrated Circuits to compute SHA-256 so the upper bound from [22] implicitly incorporates this dramatic cost reduction. By contrast, the adversary cannot (significantly) reduce the cost of evaluating a memory hard function by developing customized hardware.

[6] However, this estimate of the adversary's value could be too high because it does not account for the inherent risk of getting caught when selling/using the password

the adversary's utility would be

$$\mathbf{U}^{det}_{ADV}(B, v, k) \doteq v \sum_{i=1}^{B} p_i - \left(k \sum_{i=1}^{B} i \cdot p_i + \sum_{i=B+1}^{n} B \cdot p_i \right) .$$

The first term is the adversary's expected reward. The last term is the adversary's expected guessing cost.[7] Let $B^* = B^{det,*}_v = \arg\max_B \mathbf{U}^{det}_{ADV}(B, v, k)$ denote the adversary's utility optimizing strategy. Then the fraction of passwords cracked by a rational adversary will be

$$\mathcal{P}^{det}_{ADV,v,k} \doteq \sum_{i=1}^{B^*} p_i \ . \tag{1}$$

3 CASH Mechanism

In this section we introduce the basic CASH mechanism, while deferring until later the question of how to optimize the parameters of the mechanism.

3.1 CASH Authentication.

Observe that in traditional password authentication the costs of verifying and rejecting a password guess are symmetric. The goal of CASH is to redesign the authentication mechanism so that these costs are not symmetric. In particular, we want to ensure that the cost of rejecting an incorrect password is greater than the cost of accepting a correct password. This is a desirable property because most of the adversary's password guesses during an offline attack will be incorrect. By contrast, the authentication server will spend most of its effort authenticating legitimate users.

3.1.1 Creating an Account

Suppose that a user u registers for an account with the password $pwd_u \in \mathcal{P}$. In CASH authentication the authentication server stores the value $\left(u, s_u, k, \mathbf{H}^k(pwd_u, s_u, t_u)\right)$. As before s_u is a random salt value and k is the number of hash iterations. The key difference is that we select a random value t_u from the range $\{1, \ldots, m\}$. We stress that the value t_u is not stored on the authentication server (unlike the salt value s_u). Thus, the value t_u will not be available to an adversary who breaches the server. The account creation process is formally presented in Algorithm 1. We use the notation t_u here to emphasize that this value is chosen independently for each user u. Intuitively, the parameter t_u specifies the number of times that the authentication server needs to compute $\mathbf{H}^k$ when verifying a correct password guess using CASH.

3.1.2 Authentication

When the user u tries to authenticate using the password guess pwd'_u the authentication server first locates the record $\left(u, s_u, k, \mathbf{H}^k(pwd_u, s_u, t_u)\right)$. The authentication server then computes $\mathbf{H}^k(pwd'_u, s_u, t)$ for each value $t \in \{1, \ldots, m\}$. Authentication is successful if the hashes match for *any* value $t \in \{1, \ldots, m\}$. This is guaranteed to happen after t_u steps whenever the user's password is correct ($pwd'_u = pwd_u$), and this is highly unlikely whenever the user's password is incorrect. The authentication process is formally presented in Algorithm 2.

[7] Note that for $i \leq B$ the adversary finishes early after only i guesses if and only if the user selected password pwd_i (probability p_i). If the user selected password pwd_i with $i > B$ then the adversary will quit after B guesses.

Table 1: Notation

Term	Explanation
$\mathcal{P}$	space of passwords
n	number of passwords in $\mathcal{P}$
pwd_i	the i'th most likely password in $\mathcal{P}$
p_i	probability that a random user selects pwd_i
m	the number of evaluations of $\mathbf{H}^k$ necessary to reject an incorrect password using CASH
$t \in \{1, \ldots, m\}$	hidden running time parameter which specifies the running time of CASH when verifying an correct password. t is randomly selected during account creation.
$\tilde{p}$	a distribution over the hidden running time parameter t
$\tilde{p}_j$	the probability that the running time parameter is $t = j$
π_i	the probability of the i'th most likely tuple (pwd, t)
α	probability of seeing a correct password in a random authentication session
$\mathbf{H}$	a cryptographic hash function with $\mathbf{Cost}(\mathbf{H}) = 1$
$\mathbf{H}^k$	a cryptographic hash function with $\mathbf{Cost}(\mathbf{H}^k) = k$
$C_{SRV,\alpha}$	$mk(1-\alpha)+\alpha k \sum_{t=1}^{m} t \cdot \tilde{p}_t$, the amortized cost of a random authentication session.
C_{max}	the maximum (amortized) cost that the authentication server is willing to incur per authentication
v	adversary's true value for a cracked password
$\hat{v}$	the authentication server's estimate for v
$\mathcal{P}^{CASH}_{ADV,v,\hat{v},C}$	the fraction of passwords cracked by a rational value v adversary, when the authentication server optimizes the CASH distribution $\tilde{p}$ under the belief $\hat{v}$ subject to the cost constraint $C_{SRV,\alpha} \leq C_{max}$.
$\mathcal{P}^{pepper}_{ADV,v,C}$	the fraction of passwords cracked by a rational value v adversary, when the authentication server uses the uniform distribution $\tilde{p}_i = 1/m$. The hash cost parameter k is now tuned subject to the cost constraint $C_{SRV,\alpha} \leq C$.
$\mathcal{P}^{det}_{ADV,v,C}$	the fraction of passwords cracked by a rational value v adversary when the authentication server uses deterministic key-stretching techniques. The hash cost parameter is set to $k = C$ so that the servers cost is C for each authentication session.

3.1.3 CASH Notation

We use $\tilde{p}_i$ to denote the probability that we set $t_u = i$ during the account creation process. For notational convenience we will assume that these values are sorted so that $\tilde{p}_1 \geq \ldots \geq \tilde{p}_m$. We will use $t \leftarrow \tilde{p}$ to denote a random sample from $\{1, \ldots, m\}$ in which $\Pr_{t \leftarrow \tilde{p}}[t = i] = \tilde{p}_i$. For now we assume that the CASH distribution

$\tilde{p}$ is given to us. In later sections we will discuss how to select a good distribution $\tilde{p}$.

Algorithm 1 CASH: Create Account

Input: u, pwd_u, $\tilde{p} = (\tilde{p}_1, \ldots, \tilde{p}_m)$, k, L

1: $s_u \xleftarrow{\$} \{0,1\}^L$
2: $t_u \leftarrow \tilde{p}$
3: $h \leftarrow \mathbf{H}^k(pwd_u, s_u, t_u)$
4: **StoreRecord** (u, s_u, k, h)

Algorithm 2 CASH:Authenticate

Input: u, pwd_u

1: $R \leftarrow$ **TryFindRecord** (u)
2: **if** $R = \emptyset$ **then**
3: **return** "Username Not Found."
4: **end if**
5: $(u, s_u, k, h) \leftarrow R$
6: **for** $t = 1, \ldots, m$ **do**
7: $h_t \leftarrow \mathbf{H}^k(pwd_u, s_u, t)$
8: **if** $h_t = h$ **then**
9: **return** "Authentication Successful"
10: **end if**
11: **end for**
12: **return** "Authentication Failed"

3.2 Cost to Server

The cost of rejecting an incorrect password guess is $m \cdot k$ because the server must evaluate $\mathbf{H}^k(pwd_u, s_u, t_u)$ for all m possible values of $t_u \in \{1, \ldots, m\}$. However, whenever a password guess is correct the authentication server can halt computation as soon as it finds a match, which will happen after t_u iterations. Here, we assume that the authentication server will minimize its amortized cost by trying the most likely values of t_u first. If we let α denote the probability that the user enters his password correctly during a random authentication session then the amortized cost of the authentication server is

$$C_{SRV,\alpha} \doteq (1-\alpha)\, k \cdot m + \alpha \cdot k \sum_{i=1}^{m} i \cdot \tilde{p}_i \ .$$

In general, we will assume that the server has a maximum amortized cost C_{max} that it is willing to incur for authentication.[8] Thus, the authentication server must pick the distribution $\tilde{p}$ subject to the cost constraint $C_{SRV,\alpha} \leq C_{max}$.

3.3 Adversary Response

Fixing the CASH distribution $\tilde{p}$ induces a distribution over pairs $(pwd, t) \in \mathcal{P} \times \{1, \ldots, m\}$, namely $\Pr[(pwd, t)] = p_i \cdot \tilde{p}_t$. Once the adversary selects a threshold B the adversary's optimal strategy is to try the B most likely pairs. In this case the adversary's utility will be

$$\mathbf{U}_{ADV}^{CASH}(B, v) = v \sum_{i=1}^{B} \pi_i - k \sum_{i=1}^{B} i \cdot \pi_i - k \sum_{i=B+1}^{mn} B \cdot \pi_i \ , \tag{2}$$

[8] For example, C_{max} might be (approximately) given by the maximum computational load that the authentication server(s) can handle divided by the maximum (anticipated) number of users authenticating at any given point in time.

where the terms $\pi_1 \geq \ldots \geq \pi_{mn}$ denote the probabilities of each pair $(pwd, t) \in \mathcal{P} \times \{1, \ldots, m\}$ (in sorted order). In general, the distribution $\tilde{p}$ that the authentication server selects may depend on the maximum (amortized) server cost C_{max} as well as our belief $\hat{v}$ about the adversary's value for a cracked password. Once $\hat{v}$ and C_{max} (and thus $\tilde{p}_1, \ldots, \tilde{p}_m$, k and $\pi_1, \ldots, \pi_{mn}$) have been fixed we can let $B^* = B_v^{CASH,*} = \arg\max_B \mathbf{U}_{ADV}^{CASH}(B, v)$ denote the adversary's utility optimizing response. Then the fraction of passwords cracked by a rational adversary will be

$$\mathcal{P}_{ADV,v,\hat{v},C_{max}}^{CASH} \doteq \sum_{i=1}^{B^*} \pi_i \ . \tag{3}$$

Similarly, we will use $\mathcal{P}_{ADV,v,C_{max}}^{pepper}$ to denote the fraction of passwords cracked by a rational adversary when $\tilde{p}$ is the uniform distribution.[9] In this case the hash cost parameter k is tuned to to ensure that $C_{SRV,\alpha} \leq C_{max}$ — this can be achieved when

$$k = \frac{C_{max}}{(1-\alpha)m + \alpha\left(\frac{m+1}{2}\right)} \ . \tag{4}$$

3.3.1 Example Distribution

One simple, yet elegant, way to achieve the goal of cost asymmetry is to set $\tilde{p}_j = \frac{1}{m}$ for each $j \in \{1, \ldots, m\}$ [46]. We will sometimes call this solution uniform-CASH in this paper because it is a special case of the CASH mechanism. The amortized cost of verifying a correct password guess with uniform-CASH is $C_{SRV,1} = k\left(\frac{m+1}{2}\right)$. By contrast, the cost of rejecting an incorrect password guess is $k \cdot m$ — approximately twice the cost of verifying a correct password guess.

Examples with Analysis The above mechanism can already be used to significantly reduce the fraction of user passwords that would be cracked in an offline attack. We demonstrate the potential power of CASH with two (simplistic) examples. To keep the examples simple we will assume that that users never forget or mistype their passwords (i.e., $\alpha = 1$). In the first example, every user selects one of two passwords (e.g., pwd_1 ="123456" and pwd_2="iloveyou") with probability $p_1 = 2/3$ and $p_2 = 1/3$ respectively, and the untargeted adversary has a value of $v = \frac{4}{3}C_{max} + \epsilon$, just slightly more than C_{max} — the amortized cost incurred by the authentication server during an authentication session.

- (Deterministic Key-Stretching) The defender sets the hash cost parameter $k = C_{max}$ and stores the deterministic hash value $\mathbf{H}^k$. It is easy to check that the adversary's optimal response is to choose the maximum threshold $B^* = 2$. In this case the adversary cracks the password with probability $\mathcal{P}_{ADV,v,C_{max}}^{det} = 1$.
- (Uniform CASH) The defender sets $\tilde{p}_i = \frac{1}{m}$ for each i and he selects cost parameter $k = 2 \cdot C_{max}/(m+1)$ to ensure that $C_{SRV,1} \leq C_{max}$ — see eq 4 . It is not too difficult to see that the adversary's optimal response is to choose the threshold $B^* = 0$ (i.e., give up without guessing).[10] Thus, $\mathcal{P}_{ADV,v,C_{max}}^{pepper} = 0 < 1 = \mathcal{P}_{ADV,v,C_{max}}^{det}$.

This first example illustrates the potential advantage of randomization. The next example illustrates the potential advantage of non-uniform distributions. Example 2 is the same as example 1 except that we increase the adversary's value to $v = \frac{5}{3}C_{max}$.

- (Deterministic Key-Stretching) Increasing v can only increase $\mathcal{P}_{ADV,v,C_{max}}^{det}$. Thus, $\mathcal{P}_{ADV,v,C_{max}}^{det} = 1$.
- (Uniform CASH) Now the adversary's optimal strategy is to choose the maximum threshold $B^* = 2m$ (i.e., keep guessing until he finds the password). Thus, $\mathcal{P}_{ADV,v,C_{max}}^{pepper} = 1$.

[9] Note that $\mathcal{P}_{ADV,v,C_{max}}^{pepper}$ does not depend on $\hat{v}$, our belief about the adversary's value, because the choice of $\tilde{p}$ (and k) is independent of this belief.

[10] In particular, if the adversary instead sets $B^* = 2m$ (i.e., keep guessing until he succeeds) then his expected guessing costs will be $p_1 k\left(\frac{m+1}{2}\right) + (1-p_1)k\left(m + \frac{m+1}{2}\right) = (1-p_1)km + \frac{m+1}{2}k = C_{max} + \frac{1}{3}\left(\frac{2mC_{max}}{m+1}\right) = \frac{5C_{max}}{3} - \frac{2C_{max}}{m+1} > v$.

- (non-uniform CASH) Suppose that the authentication server, knowing that $\hat{v} = v = \frac{5 \cdot C_{max}}{2}$, sets $m = 5$, $k = C_{max}/2$ and sets $\tilde{p}_1 = 9/16, \tilde{p}_2 = \tilde{p}_3 = \tilde{p}_4 = 1/8$ and $\tilde{p}_5 = 1/16$.[11] In this case it is possible to verify that the adversary's optimal response is to set $B^* = 2$ meaning that the adversary will try guessing the two most likely pairs $(pwd_1, t = 1)$ and $(pwd_2, t = 1)$ before giving up. Thus, $\mathcal{P}^{CASH}_{ADV,v,\hat{v},C_{max}} = (p_1 + p_2)\tilde{p_1} = \frac{9}{16} < 1$.

Admittedly these example are both overly simplistic. However, we will later consider several empirical password distributions and demonstrate that non-uniform CASH distributions are often significantly better than both uniform CASH and deterministic key-stretching.

4 Stackelberg Model

In the last section we observed that uniform CASH can reduce the adversary's success rate compared to deterministic key-stretching techniques with comparable costs. We also saw that sometimes it is possible to do even better than uniform CASH by selecting a non-uniform distribution over t.[12] This observation leads us to ask the following question: What distribution over t leads to the optimal security results?

In this section we first formalize the problem of finding the optimal CASH distribution parameters $\tilde{p}_1 \geq \ldots \geq \tilde{p}_m \geq 0$. Intuitively, we can view this problem as the problem of computing the Stackelberg equilibria of a certain game between the authentication server and an untargeted offline adversary. Stackelberg games and their applications have been an active area of research in the last decade (e.g., [15, 29, 40, 59]). For now we will simply focus on formulating this goal as an optimization problem. In later sections we will present a polynomial time algorithm to good solutions to this optimization problem (Sections 5 and 5.2) and we will evaluate this algorithm on empirical password datasets (Section 6).

Before the Stackelberg game begins the adversary is given a value v for cracked passwords and the authentication server is given an honest estimate $\hat{v} = v$ of the adversary's value.[13] The authentication server is also given a bound C_{max} on the expected cost of an authentication round.

Defender Action The authentication server (leader) moves first in our Stackelberg game. The authentication server must commit to a CASH distribution $\tilde{p}$ and a hash cost parameter k. The values must be selected subject to a constraint on the maximum amortized cost for the authentication server

$$C_{SRV,\alpha} = (1 - \alpha)\, m \cdot k + \alpha \cdot k \sum_{i=1}^{m} (i \cdot \tilde{p}_i) \leq C_{max} \ .$$

Intuitively, we can view the value α as being given by nature and the parameter C_{max} is given by the computational resources of the authentication server.

Offline Adversary After the authentication server commits to $\tilde{p}$ and k the offline adversary is given access to all of the hashed passwords stored on the authentication server. The adversary can try guesses of the form (pwd_i, j). This particular guess is correct if and only if the user u selected password $pwd_u = pwd_i$ and we selected the secret salt value $t_u = j$. For an untargeted attacker the probability that this guess is correct is $p_i \cdot \tilde{p}_j$. We can describe the action of a rational adversary using a threshold B which denotes the maximum number of pairs (pwd, t) that he will check (equivalently the maximum number of times he will compute $\mathbf{H}^k$). Intuitively, we don't need to specify which pairs the adversary guesses because a rational adversary will always check the B most likely pairs.

[11] It is easy to verify that $C_{SRV,\alpha} = 2k = C_{max}$.

[12] Of course in some cases the uniform distribution might still be optimal.

[13] In the game the authentication server will assume that $\hat{v}$ is indeed the correct value when he computes the distribution $\tilde{p}$. Of course, in our empirical analysis we will also be interested in exploring how CASH performs when this estimate is incorrect $\hat{v} \neq v$.

We remark that we assume that an offline attacker will be able to obtain the CASH parameters $\tilde{p}_1, \ldots, \tilde{p}_m$ and k that we select.[14] The adversary also knows the empirical password distribution $p_1 \geq \ldots \geq p_n$ and the associated passwords $pwd_1, \ldots, pwd_n$.

Optimization Goal Informally, the defender's goal is to minimize the probability that the rational adversary succeeds in cracking each user's password. The distribution that achieves this goal is the Stackelberg equilibrium of our game. Formally, our optimization goal is presented as Optimization Goal 1. We are given as input the empirical password distribution $p_1, \ldots, p_n$ as well as the value $\hat{v}$ for the adversary, a maximum cost C_{max} for the authentication server, the CASH parameter m and the fraction α of authentication sessions in which enter their correct password. We want to find values $\tilde{p}_1, \ldots, \tilde{p}_m$ and k that minimize the fraction of cracked passwords $\mathcal{P}^{CASH}_{ADV,v,\hat{v},C_{max}}$ subject to several constraints. Constraints 1 and 2 ensure that $\tilde{p}_1, \ldots, \tilde{p}_m$ form a valid probability distribution, and constraint 3 ensures that the amortized cost of authentication is at most C_{max}. Constraint 4 simply defines the variables $\pi_1, \ldots, \pi_{mn}$ where π_i is the probability of the i'th most likely tuple (pwd, t). Constraints 5 implies that B^* is the adversary's optimal response (e.g., $\mathbf{U}^{CASH}_{ADV}(B^*, v) \geq \mathbf{U}^{CASH}_{ADV}(B, v)$ for any other threshold B that the adversary might choose). Finally, $\sum_{i=1}^{B^*} \pi_i$, our minimization goal, is the fraction of passwords cracked under the adversary's utility optimizing response B^*.

Optimization Goal 1: Minimize Adversary Success Rate

Input Parameters: $p_1, \ldots, p_n$, $\hat{v}$, C_{max}, m and α.
Variables: $\tilde{p}_1, \ldots, \tilde{p}_m, \pi_1, \ldots, \pi_{nm}$, k
minimize $\sum_{i=1}^{B^*} \pi_i$ subject to
(1) $1 \geq \tilde{p}_1 \geq \ldots \geq \tilde{p}_m \geq 0$,
(2) $\sum_{i=1}^{m} \tilde{p}_i = 1$,
(3) $(1-\alpha)\, mk + \alpha k \sum_{i=1}^{m} (i \cdot \tilde{p}_i) \leq C_{max}$,
(4) $\pi_1, \ldots, \pi_{mn} = \mathbf{Sort}(p_1 \cdot \tilde{p}_1, \ldots, p_n \cdot \tilde{p}_m)$, and
(5) $\forall B \in \{0, 1, \ldots, mn\}$ we have

$$\mathbf{U}^{CASH}_{ADV}(B^*, v) \geq \mathbf{U}^{CASH}_{ADV}(B, v) \ .$$

Unfortunately, Optimization Goal 1 is inherently non-convex due to the combination of constraints 4 and 5.[15] Thus, it is not clear whether or not there is a polynomial time algorithm to compute the Stackelberg equilibria. However, as we will see in the next section, there is a polynomial time algorithm to solve a very closely related goal. Minimize the number of passwords that a threshold B adversary can crack (Goal 2).

5 Algorithms

In this section we show how the goal of minimizing the success rate of a threshold B adversary can be formulated as a linear program with exponentially many constraints (Optimization Goal 2). We also show that this linear program can be solved in polynomial time by developing an efficient separation oracle. Unfortunately, this polynomial time algorithm is not efficient enough to solve the large real-world instances we consider in our experiments in Section 6. However, building on ideas from Section 5, we develop a more efficient (in practice) algorithm in Section 5.2. This new algorithm always finds an approximately optimal solution to Optimization Goal 2. While we do not have any theoretical guarantees about its running time, we

[14] An offline adversary has already breached authentication server which will contain code to sample t_u whenever a new user u creates an account.

[15] Substituting in the formula for $\mathbf{U}^{CASH}_{ADV}(B, v)$ constraint 5 becomes $v\sum_{i=1}^{B} \pi_i - k\sum_{i=1}^{B} i \cdot \pi_i - k\sum_{i=B+1}^{mn} B \cdot \pi_i \leq v\sum_{i=1}^{B} \pi_i - k\sum_{i=1}^{B^*} i \cdot \pi_i - k\sum_{i=B^*+1}^{mn} B^* \cdot \pi_i$, where π_i depends on the **Sort** operation.

found that it converged quickly on every instance we tried. Furthermore, as we will see in our experimental evaluation, the algorithm results in significantly improved Stackelberg strategies.

We remark that our experimental results in Section 6 can be understood without reading this section. In particular, it is possible to view the algorithms in Sections 5 and 5.2, as a blackbox heuristic algorithm that finds reasonably good solutions to Optimization Goal 1. A more empirically inclined reader may wish to skip to our experimental results in Section 6 after skimming through this section.

5.1 LP Formulation

We first show how to state our goal, minimize the number of passwords that a threshold B adversary will crack, as a linear program. Our LP uses the following variables $\mathcal{P}_{Adv,B}, \tilde{p}_1, \ldots, \tilde{p}_m$. Intuitively, the variable $\mathcal{P}_{Adv,B}$ represents the fraction of passwords that a threshold B adversary can crack. At a high level our Linear Program can be understood as follows: minimize $\mathcal{P}_{Adv,B}$ subject to the requirement that no feasible strategy for the threshold B adversary achieves a success rate greater than $\mathcal{P}_{Adv,B}$. This requirement can be expressed as a combination of exponentially many linear constraints. Formally, our LP is presented as Optimization Goal 2.

Optimization Goal 2: Minimize Threshold B Adversary Success Rate

Input Parameters: $p_1, \ldots, p_n$, B, C_{max}, m, k, α
Variables: $\tilde{p}_1, \ldots, \tilde{p}_m, \mathcal{P}_{Adv,B}$
minimize $\mathcal{P}_{Adv,B}$ subject to
(1) $1 \geq \tilde{p}_1 \geq \ldots \geq \tilde{p}_m \geq 0$,
(2) $\sum_{i=1}^{m} \tilde{p}_i = 1$,
(3) $(1-\alpha)\, mk + \alpha k \sum_{i=1}^{m} (i \cdot \tilde{p}_i) \leq C_{max}$,
(4) $0 \leq \mathcal{P}_{Adv,B} \leq 1$, and
(5) $\forall S \subset \mathcal{P} \times \{1, \ldots, m\}$ s.t $|S| = B$ we have

$$\mathcal{P}_{Adv,B} \geq \sum_{(i,j) \in S} p_i \cdot \tilde{p}_j \ .$$

The key intuition is that all of the (5) constraints ensure that $\mathcal{P}_{Adv,B}$ is at least at big as the best success rate for a threshold B adversary. This is true because the optimal guessing strategy for a threshold B adversary is to guess the B most likely tuples (pwd, t). Let S' denote these B most-likely tuples then one of the type (5) constraints says that $\mathcal{P}_{Adv,B} \geq \sum_{(i,j) \in S'} p_i \cdot \tilde{p}_j$. Thus, type (5) constraints guarantee we cannot 'cheat' by pretending like the adversary will follow a suboptimal strategy (e.g., spending his guessing budget on the least likely passwords) when we solve Optimization Goal 2.

The key challenge in solving Optimization Goal 2 is that there are exponentially many type (5) constraints. Our main result in this section states that we can still solve this problem in polynomial time.

Theorem 1. *We can find the solutions to Optimization Goal 2 in polynomial time in m, n and L, where L is the bit precision of our inputs.*

The proof of Theorem 1 can be found in the appendix. We briefly overview the proof strategy here. The key idea is to build a polynomial time separation oracle for Optimization Goal 2. Given a candidate solution $\tilde{p}$ the separation oracle should either tell us that the solution is feasible (satisfies all type (5) constraints) or it should find an unsatisfied constraint. We can then use the ellipsoid method [44] with our separation oracle to solve to solve the linear program in polynomial time. In appendix .1 we show how to develop a polynomial time separation oracle for our linear programs. Intuitively, the separation oracle simply sorts the tuples $\mathcal{P} \times \{1, \ldots, m\}$ using the associated probabilities $\Pr[(pwd_i, t)] = p_i \cdot \tilde{p}_t$. Then we can find the set S' of the B most likely tuples and check to see if the constraint $\mathcal{P}_{Adv,B} \geq \sum_{(i,j) \in S'} p_i \cdot \tilde{p}_j$ is satisfied.

Once we have a polynomial time algorithm to solve Optimization Goal 2 for a fixed value of k we could adopt the multiple LP framework of Conitzer and Sandholm [29] to include k as an optimization parameter. The idea is simple. Because the range of possible values of k is small ($k \leq C_{max}$) we can simply solve Optimization Goal 2 separately for each value of k and take the best solution — the one with the smallest value of $\mathcal{P}_{Adv,B}$.

5.2 Practical CASH Optimization

Theorem 1 states that Optimization Goal 2 can be solved in polynomial time using the ellipsoid algorithm [44]. While this is nice in theory the ellipsoid algorithm is rarely deployed in practice because the running time tends to be very large. In this section we develop a heuristic algorithm (Algorithm 3) to solve Goal 2 using our separation oracle. While algorithm 3 is guaranteed to always find the (approximately) optimal solution to Optimization Goal 2, we do not have any theoretical proof that it will converge to find the optimal solution in polynomial time. However, in all of our experiments we found that Algorithm 3 converged reasonably quickly.

The basic idea behind our heuristic algorithm is to start by ignoring all of the type (5) constraints from Goal 2. We then run a standard LP solver to find the optimal solution to the resulting LP. Finally, we run our separation oracle to determine if this solution violates any type (5) constraints. If it does not then we are done. If the separation oracle does find a violated type (5) constraint then we add this constraint to our LP and solve again. We repeat this process until we have a solution that satisfies all type (5) constraints. Observe that this process must terminate because we will eventually run out of type (5) constraints to add. The hope is that our algorithm will converge much more quickly. In practice, we find that it does (e.g., at most 25 iterations).

Further Optimizations Our separation oracle runs in time $O(mn \log mn)$ because we sort a list of mn tuples (pwd, t). In practice, the number of passwords n might be very large (e.g., the RockYou dataset contains $n \approx 14.3 \times 10^6$ unique passwords). Fortunately, it is often possible to drastically reduce the time and space requirements of our separation oracle by grouping passwords into equivalence classes. In particular, we group two passwords pwd_i and pwd_j into an equivalence class if and only if $p_i = p_j$. This approach reduces running time of our separation oracle to $O(mn' \log mn')$, where n' is the number of equivalence classes[16]. For example, the RockYou database contains over 10^7 unique passwords, but we only get $n' = 2040$ equivalence classes.

We can represent our empirical distribution over passwords as a sequence of n' pairs $(p_1, n_1), \ldots, (p_{n'}, n_{n'})$. Here, p_i denotes the probability of a password in equivalence class i and $n_i \in \mathbb{N}$ denotes the total number of passwords in equivalence class i. We have $\sum_{i=1}^{n'} n_i = n$ and $\sum_{i=1}^{n'} n_i \cdot p_i = 1$. As before we assume that $p_i \geq p_{i+1}$. In most password datasets $n_{n'}$ is the number of passwords that were selected by only one user (e.g., for the RockYou dataset $n_{n'} \approx 11.9 \times 10^6$).

We now argue that this change in view does not fundamentally alter our linear program (Optimization Goal 2) or our separation oracle. Constraints (1)–(4) in our LP remain unchanged. We need to make a few notational changes to type (5) constraints to ensure that $\mathcal{P}_{Adv,B}$ is at least as large at the success rate of the optimal adversary. We use

$$\mathcal{F}_B = \left\{ (b_1, \ldots, b_{n'}) \in \mathbb{N}^{n'} \left| \sum_{i=1}^{n'} b_i \leq B \wedge \forall i. b_i \leq m \cdot n_i \right. \right\},$$

to describe the space of feasible guessing strategies for an adversary with a threshold B. Here, b_i denotes the total number of times the adversary evaluates $\mathbf{H}^k$ while attacking passwords in equivalence class $i \leq n'$. Thus, the range of b_i is $0 \leq b_i \leq m \cdot n_i$ because there are n_i passwords in the equivalence class to attack and he can choose to evaluate $\mathbf{H}^k$ up to m times for each password.

[16] To save computation one could also group passwords into equivalence classes with *approximately* equal probabilities, but this representation loses some accuracy and was unnecessary in all of our experiments.

Given values $\tilde{p}_1, \ldots, \tilde{p}_m$ and a feasible allocation $b_1, \ldots, b_{n'} \in \mathcal{F}_B$ the probability that adversary will crack the password is at most

$$\sum_{i=1}^{n'} p_i \left((b_i \mod n_i) \, \tilde{p}_{\lceil \frac{b_i}{n_i} \rceil} + \sum_{j=1}^{\lfloor \frac{b_i}{n_i} \rfloor} n_i \tilde{p}_j \right) .$$

Intuitively, the optimal adversary will spend equal effort (b_i/n_i) cracking each password in an equivalence class because they all have the same probability. The $(b_i \mod n_i)$ and $\lfloor b_i/n_i \rfloor$ terms handle the technicality that b_i may not be divisible by n_i. Thus, we can replace our type (5) constraints with the constraint

$$\mathcal{P}_{Adv,B} \geq \sum_{i=1}^{n'} p_i \left((b_i \mod n_i) \, \tilde{p}_{\lceil \frac{b_i}{n_i} \rceil} + \sum_{j=1}^{\lfloor \frac{b_i}{n_i} \rfloor} n_i \tilde{p}_j \right) ,$$

for every $(b_1, \ldots, b_{n'}) \in \mathcal{F}_B$.

Our modified separation oracle works in essentially the same way. We sort the tuples (i, j) using the values $p'_{i,j} = p_i \cdot \tilde{p}_j$ and select the B largest tuples. The only difference is that the adversary is now allowed to select the tuple (i, j) up to n_i times. In this section we will use **SeparationOracle** to refer to the modified separation oracle, which runs in time $O(mn' \log mn')$ using our compact representation of the empirical password distribution.

Our heuristic algorithm (Algorithm 3) takes as input an approximation parameter ϵ. It is allowed to output a solution $\tilde{p}_1, \ldots, \tilde{p}_m, \mathcal{P}_{Adv,B}$ as long as the solution is within ϵ of optimal — for any other feasible solution $\tilde{p}'_1, \ldots, \tilde{p}'_m, \mathcal{P}'_{Adv,B}$ we have $\mathcal{P}_{Adv,B} \leq \mathcal{P}'_{Adv,B} + \epsilon$. We use **Slack** to denote a function that computes how badly a linear inequality C is violated. For example, if C denotes the inequality $x + y \geq 2.5$ and we have set $x' = y' = 1$ then $\textbf{Slack}(C, x', y') = 0.5$ (e.g., if we introduced a slack variable z then we would need to select z' such that $|z'| = 0.5$ to satisfy the inequality $x' + y' + z' \geq 2.5$).

Algorithm 3 Optimize $(p, n, B, C_{max}, \alpha, \epsilon, m, S)$

Input: $p_1, \ldots, p_{n'}$, $n_1, \ldots, n_{n'}$, B, C_{max}, α, ϵ, m, $S = \{k_0, k_1, \ldots, , k_\tau\}$,

$bestSolution \leftarrow \emptyset$, $bestK \leftarrow k_0$
$bestSuccessRate \leftarrow 1.0$, $slack \leftarrow \epsilon$
for $j = 0, \ldots, \tau$ **do**
  $k \leftarrow k_j$
  $C \leftarrow$ **InitialConstraints**(C_{max}, α, k)
  {Initially, C only includes constraints (1)–(4)
  in goal 2}
  $Goal \leftarrow \{\min \mathcal{P}_{Adv,B}\}$
  $Vrbls \leftarrow \{\mathcal{P}_{Adv,B}, \tilde{p}_1, \ldots, \tilde{p}_m\}$
  $\mathcal{P}'_{Adv,B}, \tilde{p}'_1, \ldots, \tilde{p}'_m \leftarrow$ **LPSolve** $(Goal, Vrbls, C)$
  $\tilde{p}' \leftarrow (\tilde{p}'_1, \ldots, \tilde{p}'_m)$
  $\vec{p} \leftarrow (p_1, \ldots, p_{n'})$
  $\vec{n} \leftarrow (n_1, \ldots, n_{n'})$
  $Sep_{in} \leftarrow \left(\vec{p}, \vec{n}, \tilde{p}', B, k, C_{SRV,\alpha}, \mathcal{P}'_{Adv,B}\right)$
  $C' \leftarrow$ **SeparationOracle** (Sep_{in})
  while $\left|\textbf{Slack}\left(C', \tilde{p}, \mathcal{P}'_{Adv,B}\right)\right| > \epsilon \wedge (C' \neq$ "Ok") **do**
    $C \leftarrow C \bigcup \{C'\}$
    $\mathcal{P}'_{Adv,B}, \tilde{p}' \leftarrow$ **LPSolve** $(Goal, Vrbls, C)$
      $\{\tilde{p}' = (\tilde{p}'_1, \ldots, \tilde{p}'_m)\}$
    $Sep_{in} \leftarrow \left(\vec{p}, \vec{n}, \tilde{p}', B, k, C_{SRV,\alpha}, \mathcal{P}'_{Adv,B}\right)$
    $C' \leftarrow$ **SeparationOracle** (Sep_{in})
  end while
  if $bestSuccessRate \geq \mathcal{P}'_{Adv,B}$ **then**
    $bestSolution \leftarrow \tilde{p}_1, \ldots, \tilde{p}_m$
    $bestSuccessRate \leftarrow \mathcal{P}'_{Adv,B}$
    $(bestM, bestK) \leftarrow (m_i, k_i)$
    $slack \leftarrow$ **Slack** $\left(C', \tilde{p}, \mathcal{P}'_{Adv,B}\right)$
  end if
end for
return $\tilde{p}_1, \ldots, \tilde{p}_m, bestK$

5.3 Choosing a CASH Distribution

While Algorithm 3 efficiently solves optimization Goal 2, it may not yield the optimal distribution for our original Stackelberg game. In particular, while Algorithm 3 gives the optimal distribution against a threshold-B adversary, the rational adversary might choose to use a different threshold $B^* \neq B$.

We introduce a heuristic algorithm to find good Stackelberg strategies (CASH distributions) for the defender. Algorithm 4 uses Algorithm 3 as a subroutine to search for good CASH distributions. Algorithm 4 takes as input an (estimate) $\hat{v}$ of the adversary's value and a set $\mathcal{B}$ of potential adversary thresholds B and runs Algorithm 3 to compute the optimal distribution for each threshold. We then compute the rational value $\hat{v}$ adversary's best response to each of distributions and find the best distribution for the authentication server — the one which results in the lowest fraction of cracked passwords under the corresponding best adversary response. Algorithm 4 assumes a subroutine **RationalAdvSuccess** $(p, n, \hat{v}, \tilde{p}, k)$, which computes the fraction of cracked passwords under a value $\hat{v}$ adverary's best response to the CASH distribution $\tilde{p}$ with empirical password distribution defined by the pair (p, n) and a hash cost parameter k.

Algorithm 4 FindCASHDistribution

Input: $p_1, \ldots, p_{n'}$, $n_1, \ldots, n_{n'}$, $\hat{v}$, C_{max}, α, ϵ, m, $S = \{k_0, k_1, \ldots, k_\tau\}$, $\mathcal{B} = \{B_0, B_1, \ldots, B_\ell\}$

$\tilde{p}_1, \ldots, \tilde{p}_m \leftarrow 1/m$

$$k \leftarrow \frac{C_{max}}{(1-\alpha)m + \alpha\left(\frac{m+1}{2}\right)}$$

$advSuccess \leftarrow \mathcal{P}^{pepper}_{ADV,\hat{v},C_{max}}$
for $x = 0, \ldots, \ell$ **do**
  $B \leftarrow B_x$
  $\tilde{p}_x, k_x \leftarrow$ **Optimize** $(p, n, B, C_{max}, \alpha, \epsilon, m, S)$
  $CS \leftarrow$ **RationalAdvSuccess** $(p, n, \hat{v}, \tilde{p}_x, k_x)$
  if $CS \leq advSuccess$ **then**
    $\tilde{p} \leftarrow \tilde{p}_x$
    $k \leftarrow k_x$
    $advSuccess \leftarrow CS$
  end if
end for
return $\tilde{p}$, k

We remark that the subroutine **RationalAdvSuccess** can be computed in time $O\big(n'm \log mn'\big)$ — the most expensive step is sorting the mn' pairs $(p_i, \tilde{p}_j)$ based on the value $p_i \cdot \tilde{p}_j$. Once we have these pairs in sorted order there is a simple formula for computing the marginal benefit/costs of a larger threshold B. See Algorithm 7 in the appendix for more details.

We remark that Algorithm 4 is not guaranteed to always find the optimal solution to optimization goal 1. It may be viewed as a heuristic algorithm that generates many promising candidate CASH distributions and then selects the best distribution among them.

6 Experimental Results

In this section we empirically demonstrate that our CASH mechanism can be used to significantly reduce the fraction of accounts that an offline adversary could compromise. We implemented Algorithm 4 in C# using Gurobi as our LP solver, and analyzed CASH using two real-world password distributions $p_1, \ldots, p_n$. The first distribution is based on data from the RockYou password breach (32+ million passwords) and the second is based on password frequency data from Yahoo! users (representing ≈ 70 million passwords). The later dataset was not the result of a security breach. Instead, Yahoo! gave Bonneau [20] permission to collect and analyze password frequency data in a carefully controlled environment. Yahoo! recently allowed Blocki et al. [16] to use a differentially private [34] algorithm to publish this data. Thus, the password frequency data in this data set has been perturbed slightly. Blocki et al. [16] also showed that with high probability the L1 error introduced by their algorithm would be minimal.

In each of our experiments we fix the password correctness rate $\alpha \in \{1, 0.95, 0.9\}$ and the maximum amortized server cost C_{max} before using Algorithm 4 to find a CASH parameters $\tilde{p}_1, \ldots, \tilde{p}_m$ and k subject to the appropriate constraints on the amortized server costs.

We compare the % of cracked passwords under three different scenarios:

- (Deterministic Key-Stretching) The authentication server selects a hash function $\mathbf{H}^k$ with cost parameter $k = C_{max}$ (achieved through traditional deterministic key-stretching techniques). The rational value v adversary will crack each password with probability $\mathcal{P}^{det}_{ADV,v,k}$ (eq 1).
- (Uniform-CASH) The authentication server uses CASH with the uniform distribution. He sets k according to eq 4 to ensure that his amortized costs are at most C_{max}. A rational value v adversary will crack each password with probability $\mathcal{P}^{pepper}_{ADV,v,C_{max}}$.

- (CASH) Given an estimate $\hat{v}$ of the adversary's budget we used Algorithm 4 to optimize the CASH parameters k and $\tilde{p}_1, \ldots, \tilde{p}_m$ subject to the constraint that the amortized server cost is at most C_{max} when users enter the wrong password with probability $1-\alpha$. We fixed the parameters $m = 50, \epsilon = 0.02$, and we set $\mathcal{B} = \{5 \cdot C_{max} \times 10^4, C_{max} \times 10^6, C_{max} \times 10^7, 1.5 \cdot C_{max} \times 10^7, 2.0 \cdot C_{max} \times 10^7, 2.5 \cdot C_{max} \times 10^7, 2.65 \cdot C_{max} \times 10^7, 2.8 \cdot C_{max} \times 10^7, 3.0 \cdot C_{max} \times 10^7, 5.0 \cdot C_{max} \times 10^7, C_{max} \times 10^8\}$. Thus, Algorithm 4 computes the optimal distribution against a threshold B adversary for each $B \in \mathcal{B}$, and selects the best distribution $\tilde{p}$ against a value $\hat{v}$ adversary. $\mathcal{P}^{CASH}_{ADV,\hat{v},\hat{v},C_{max}}$ will denote the fraction of cracked passwords when the true value is $v = \hat{v}$. When the adversary's true value is $v \neq \hat{v}$, $\mathcal{P}^{CASH}_{ADV,v,\hat{v},C_{max}}$ will denote the fraction of cracked passwords.

Our results indicate that an authentication server could significantly reduce the fraction of compromised passwords in an offline attack by adopting our optimal CASH mechanism instead of deterministic key-stretching or uniform-CASH. These results held robustly for both the RockYou and Yahoo! password distributions.

6.0.1 Password Datasets

We use two password frequency datasets, RockYou and Yahoo!, to analyze our CASH mechanism. The RockYou dataset contains passwords from $N \approx 32.6$ million RockYou users, and the Yahoo! dataset contains data from $N \approx 70$ million Yahoo! users. We used frequency data from each of these datasets to obtain an empirical password distribution $p_1 \geq p_2 \geq p_3 \ldots \geq p_n$ over $\mathcal{P}$.

The RockYou dataset is based on actual user passwords which were leaked during the infamous RockYou security breach (RockYou had been storing these passwords in the clear). The total number of unique passwords in the dataset was $n \approx 14.3$ million. Approximately, 11.9 million of these passwords were unique to one RockYou user. The other ≈ 2.5 million passwords were used by multiple users. The most popular password (pwd_1 = '123456') was shared by ≈ 0.3 million RockYou users ($p_1 \approx 0.01$). RockYou did not impose strict password restrictions on its users (e.g. users were allowed to select passwords consisting of only lowercase letters or only numbers).

We also used (perturbed) password frequency data from a dataset of $N \approx 70$ million Yahoo! passwords. See [20] for more details about how this data was collected and see [16] for more details about how the frequency data was perturbed to satisfy the rigorous notion of differentially privacy [34]. Blocki et al. [16] proved that with high probability the L1 distortion of the perturbed frequency data is bounded by $O\left(\sqrt{N}/\epsilon\right)$, where the privacy parameter was set to $\epsilon = 0.25$ when the Yahoo! dataset was published. Thus, the perturbed dataset will also still give us a good estimate of the empirical password distribution.

6.1 Results

Our first set of experimental results are shown in Figures 1 and 2. These plots were computed under the assumption that $\alpha = 1$ (users always enter their passwords correctly), and that $\hat{v} = v$ (the defender knows the exact adversary value). The results show that for some (higher) adversary values our non-uniform CASH distributions improves significantly on the cost-equivalent versions of uniform CASH (50% reduction in cracked passwords) and deterministic key-stretching (56% reduction in cracked passwords).[17] Figures 4a and 4b (resp. Figures 3a and 3d) show the same results under the assumptions that $\alpha = 0.9$ (resp. $\alpha = 0.95$).

Figures 3b and 3c (resp. Figures 3f and 3e) explore the effect of a wrong estimate $\hat{v} \neq v$ of the adversary's value for both the RockYou and Yahoo! datasets. Despite receiving the wrong estimate $\hat{v}$ our algorithm returns a distribution that is (almost always) slightly better than the corresponding uniform CASH distribution. Both distributions still significantly outperform the cost equivalent deterministic key-stretching solution.

Figures 5 and 6 in the appendix explore what happens when the defender uses the wrong empirical password distribution when searching for a good CASH distribution $\tilde{p}$ (e.g., if the defender optimizes $\tilde{p}$

[17] We note that we would expect to see relatively high adversary values v/C_{max} in the offline setting because C_{max} will typically be quite small (e.g., $\$10^{-6}$).

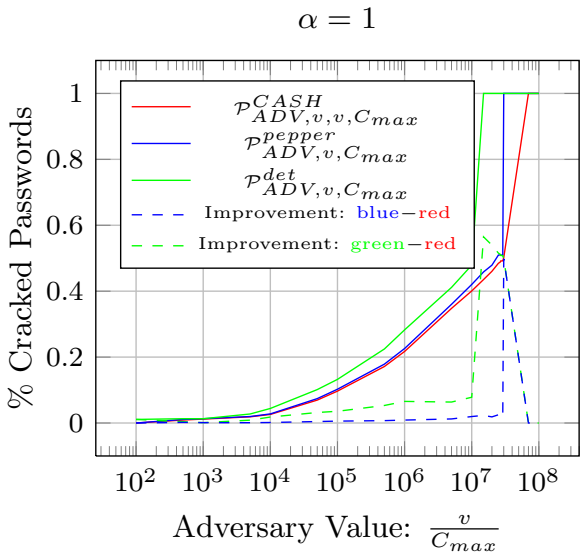


Figure 1: Yahoo Dataset: $\alpha = 1$.

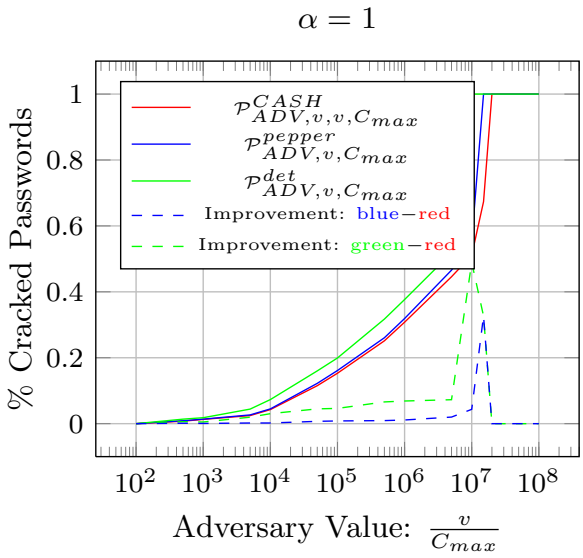
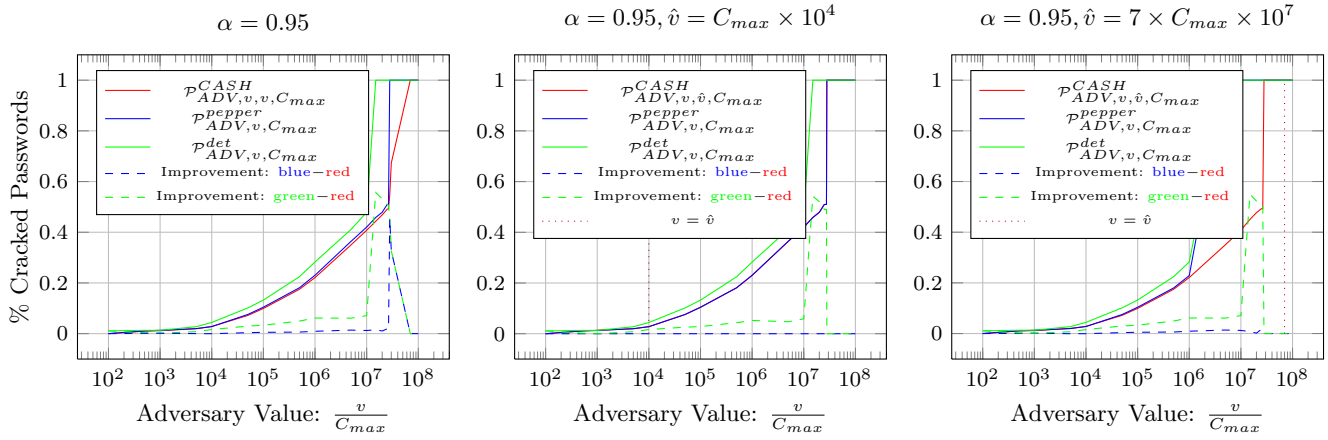
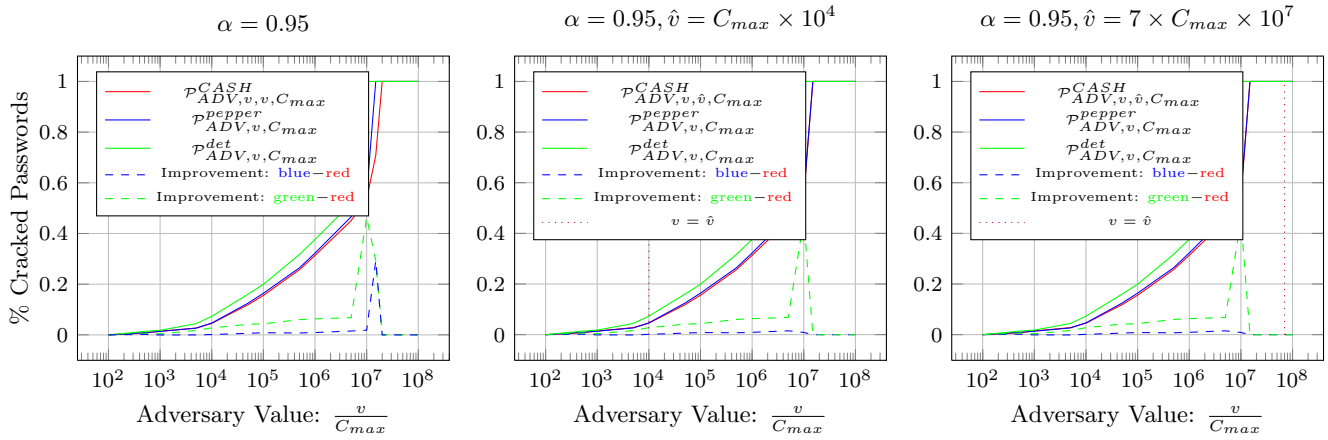


Figure 2: RockYou Dataset: $\alpha = 1$.

under the assumption that the empirical password distribution is given by the Yahoo! dataset when the actual distribution is given by the RockYou dataset). Briefly, these plots show that non-uniform CASH significantly outperforms deterministic key-stretching even when non-uniform CASH is optimized under the wrong distribution and non-uniform CASH slightly outperforms uniform CASH on most, but not all, of the curve.



(a) Yahoo (b) Yahoo: $\hat{v} \neq v$ (c) Yahoo: $\hat{v} \neq v$



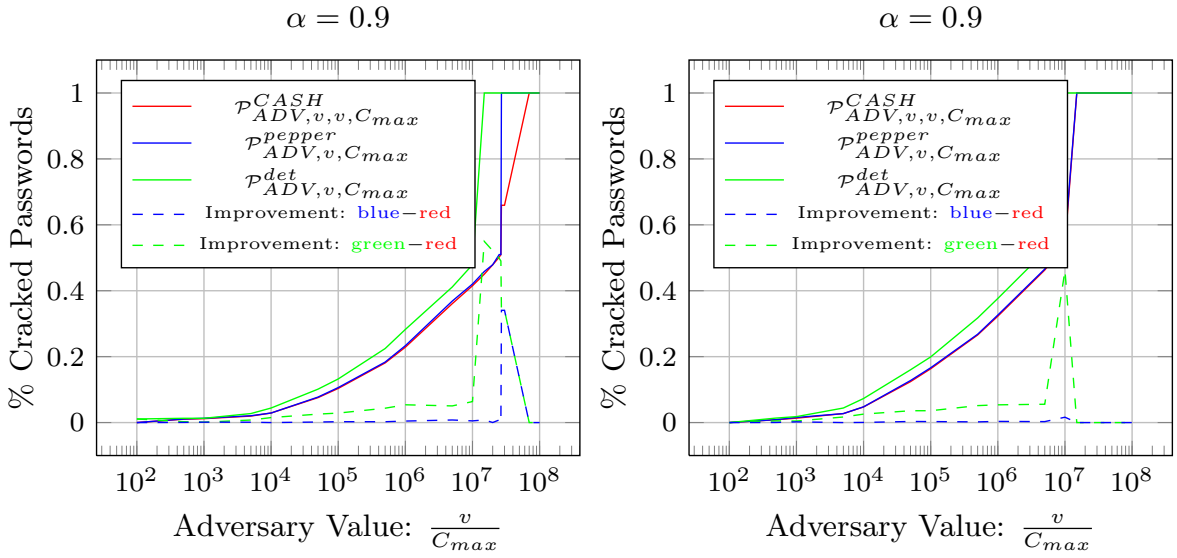
(d) RockYou (e) RockYou: $\hat{v} \neq v$ (f) RockYou: $\hat{v} \neq v$.

Figure 3: $\alpha = 0.95$

6.2 Discussion

In our experiments we varied the password correctness rate $\alpha \in \{0.9, 0.95, 1\}$. Intuitively, we expect for CASH to have a greater advantage over traditional key-stretching techniques when α is larger, but when $\alpha \to 0$ we should not expect for CASH or uniform-CASH to outperform deterministic key-stretching techniques because there is no advantage in making authentication costs asymmetric. It is easier for users to remember passwords that they use frequently [17, 22, 52] so we would expect for α to be larger for services that are used frequently (e.g., e-mail). This suggests that larger values of α (e.g., $\alpha = 0.9$ or $\alpha = 0.95$) would be appropriate for many services because the users who authenticate most frequently would be the least likely to enter incorrect passwords. While different authentication servers might experience different failed login rates $1 - \alpha$, we remark that it is reasonable to assume that the authentication server knows the value of α because it can monitor login attempts.

Estimating v While our results suggest that CASH continues to perform well even if our estimate $\hat{v}$ of the adversary's value v for cracked passwords is wrong, we would still recommend that an authentication server perform a careful economic analysis to obtain the estimate $\hat{v}$ before running Algorithm 4 to compute



(a) Yahoo Dataset.

(b) RockYou Dataset.

Figure 4: $\alpha = 0.9$.

the CASH distribution $\tilde{p}$. The organization should take into account empirical data on the cost **Cost** (**H**) of computing the underlying hash function as well as the market value of a cracked password. If possible, we recommend that the organization consider data from black market sales of passwords for similar types of accounts (e.g., an adversary would likely value a cracked Bank of America password more than a cracked Twitter password). Symantec reports that cracked passwords are sold on the black market for \$4–\$30 [36]. Thus, \$30/**Cost** (**H**) might be a reasonable upper bound on the adversary's value for a cracked password (measured in # of computations of $\mathbf{H}^k$). We would also strongly advocate for the use of memory hard functions instead of hash iteration to increase **Cost** (**H**) effectively (see discussion in Section 2.2).

Obtaining an Empirical Password Distribution We remark that the specific CASH distributions we computed for the RockYou and Yahoo! datasets might not be optimal in other application settings because the underlying password distribution may vary across different contexts. For example, users might be more motivated to pick strong passwords for higher value accounts (e.g., bank accounts). Similarly, some organizations choose to restrict the passwords that a user may select (e.g., requiring upper and lower case letters). While these restrictions do not always result in stronger passwords [45], they can alter the underlying password distribution [18]. While the underlying distribution may vary from context to context, we note that an authentication server could always follow the framework of Bonneau [20] and Blocki et al. [16] to securely approximate the password distribution $p_1, \ldots, p_n$ of its own users.

If an organization remains highly uncertain about value v of a cracked password or about the empirical password distribution $p_1, \ldots, p_n$ then it may be prudent to adopt the uniform-CASH mechanism (e.g., [46]), which *always* performs at least as well as the traditional key-stretching approach.

6.2.1 Experimental Limitations

We remark that values of $\mathcal{P}^{CASH}_{ADV,v,\hat{v},C}$ that we compute in our experiments may be less realistic for larger values of $\frac{v}{C_{SRV,\alpha}}$ (e.g., 10^8). The reason is that p_i, our empirical estimate of the probability of password pwd_i, will be too high for many of our unique passwords in the dataset. For example, consider a dedicated user who memorizes a truly random 20 character string of upper and lower case letters. The true probability

that any individual password guess matches the user's password would be at most $1/52^{20} \approx 1/\left(2.09 \times 10^{34}\right)$. However, if that password occurred in the RockYou dataset then our empirical estimate of this probability would be at least $1/\left(3.26 \times 10^{7}\right)$. Developing improved techniques for estimating the true likelihood of unique password in a password frequency dataset is an important research direction.

7 Related Work

Breaches. Recent breaches [2–8] highlight the importance of proper password storage. In one of these instances [2] passwords were stored on the authentication server in cleartext and in other instances the passwords were not salted [5]. Salting is a simple, yet effective, way to defend against rainbow table attacks [9], which can be used to dramatically reduce the cost of an offline attack against unsalted passwords [49]. Bonneau and Preibusch [21] found that implementation errors like these are unfortunately commonplace.

Key Stretching. The practice of key stretching was proposed as early as 1979 by Morris and Thompson [48]. The goal is to make the hash function more expensive to evaluate so that an offline attack is more expensive for the adversary. PBKDF2 [43], BCRYPT [54] use hash iteration to accomplish this goal. The recent Ashley Madison breach highlights the benefits of key-stretching in practice. Through an implementation mistake half of the Ashley Madison passwords were protected with the MD5 hash function instead of the much stronger BCRYPT hash function allowing offline password crackers to quickly recover these passwords[18].

More modern password hash functions like SCRYPT [51] use memory hard functions for key-stretching. Recently, the Password Hashing Competition [1] was developed to encourage the development of alternative password hashing schemes (e.g., [10, 35]). Argon2 [12], the winner, has a parameters which control memory usage and parallelism. Deterministic key-stretching methods result in proportionally increased costs for the legitimate server as well as the adversary. Manber [46] proposed the use of hidden salt values (e.g., 'pepper') to make it more expensive to reject incorrect passwords. CASH may be viewed as a generalization of this idea. Boyen [23] proposed using halting puzzles to introduce an extreme asymmetry — the password verification algorithm never halts when we try an incorrect password. However, in practice an authentication server will need to upper bound the maximum running time for authentication because even legitimate users may occasionally enter the wrong password.

Other Defenses Against Offline Attacks. If an organization has multiple servers for authentication then it is possible to distribute storage of the passwords across multiple servers to keep them safe from an adversary who only breaches one server (e.g., see [25] or [27]). Juels and Rivest [42] proposed storing the hashes of fake passwords (honeywords) and using a second auxiliary server to detect an offline attack (authentication attempt with honeywords). Another line of research has sought to include the solution(s) to hard artificial intelligence problems in the password hash so that an offline attacker needs human assistance [13, 28, 30]. By contrast, CASH does not require an organization to purchase and maintain multiple (distributed) authentication servers and it could be adopted without altering the user's authentication experience (e.g., by requiring the user to solve CAPTCHAs).

Measuring Password Strength. Guessing-entropy [47, 57], $\sum_{i=1}^{n} i \times p_i$, measures the average number of guesses needed to crack a single password. We use a similar formula to compute how much work a threshold-B adversary would do in expectation. Guessing-entropy and Shannon-entropy are known to be poor metrics for measuring password strength[19] While minimum entropy, $H_\infty = -\log p_1$, can be used to estimate the fraction of passwords that could be cracked in an online attack [18], it can provide an overly pessimistic security measurement in general.

Boztas [24] proposed a metric called β-guesswork, which measured the success rate for an adversary with β guesses per account $\sum_{i=1}^{\beta} p_i$. We use a similar formula for computing the success rate of a threshold-B adversary against our CASH mechanism — the key difference is that the adversary must guess the random value t_u as well as the user's password pwd_u. Pliam's proposed a similar metric called α-guesswork [53],

[18] See, `http://arstechnica.com/security/2015/09/once-seen-as-bulletproof-11-million-ashley-madison-passwords-already-cracked/` (retrieved 5/4/2016)

[19] Guessing-entropy could be high even if half of our users choose the same password ($p_1 = 0.5$) as long as the other half of our users choose a password uniformly at random from $\mathcal{P}\left(p_2 = \ldots = p_n = \frac{2}{n-1}\right)$.

which measures the number of password guesses the adversary would need (per user) to achieve success rate α.

Encouraging Users to Memorize Stronger Passwords. A separate line of research has focused on helping users memorize stronger passwords using various mnemonic techniques and/or rehearsal techniques (e.g., [14, 22, 39, 58]).

Password managers seek to minimize user burden by using a single password to generate multiple passwords [55]. These password managers often use client-side key stretching to derive each password. While CASH is a useful tool for server-side key stretching, our current version of CASH is not appropriate for client-side key stretching because the authentication procedure is not deterministic. In subsequent work, Blocki and Sridhar [19] developed Client-CASH an extension of CASH suitable for client-side key stretching.

Password Alternatives. Another line of research has focused on developing alternatives to text passwords like graphical passwords [11, 26, 41]. Herley and van Oorschoot argued that text passwords will remain the dominant means of authentication despite attempts to replace them [38]. We note that CASH could be used to protect graphical passwords as well as text passwords.

8 Conclusions

We presented a novel Stackelberg game model which captures the essential elements of the interaction between an authentication server (leader) and an offline password cracker (follower). Our Stackelberg model can provide guidance for the authentication server by providing an estimate of how significantly key-stretching reduces the number of passwords that would be cracked by a rational offline adversary in the event of a server breach. We also introduced, CASH, a randomized secure hashing algorithm that significantly outperforms traditional key-stretching defenses in our Stackelberg game. While the problem of computing an exact Stackelberg equlibria is non-convex, we were able to find an efficient heuristic algorithm to compute good strategies for the authentication server. Our heuristic algorithm is based on a highly related problem that we are able to show is tractable. Finally, we analyzed the performance of our CASH mechanism using empirical password data from two large scale password frequency datasets: Yahoo! and RockYou. Our empirical analysis demonstrates that the CASH mechanism can significantly (e.g., 50%) reduce the fraction of passwords that would be cracked in an offline attack by a rational adversary. Thus, our CASH mechanism can be used to mitigate the threat of offline attacks without increasing computation costs for a legitimate authentication server.

Acknowledgments

This work was completed in part while the first author was visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. The research was also supported by an AFOSR MURI on Science of Cybersecurity as well as grants from the NSF TRUST Center. The views expressed in this paper are those of the authors and do not necessarily reflect the views of the Simons Institute or the National Science Foundation.

References

[1] "Password hashing competition," https://password-hashing.net/.

[2] "Rockyou hack: From bad to worse," http://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/, December 2009, retrieved 9/27/2012.

[3] "Update on playstation network/qriocity services," http://blog.us.playstation.com/2011/04/22/update-on-playstation-network-qriocity-services/, April 2011, retrieved 5/22/2012.

[4] "Data breach at ieee.org: 100k plaintext passwords," http://ieeelog.com/, September 2012, retrieved 9/27/2012.

[5] "An update on linkedin member passwords compromised," http://blog.linkedin.com/2012/06/06/linkedin-member-passwords-compromised/, June 2012, retrieved 9/27/2012.

[6] "Zappos customer accounts breached," http://www.usatoday.com/tech/news/story/2012-01-16/mark-smith-zappos-breach-tips/52593484/1, January 2012, retrieved 5/22/2012.

[7] "Important customer security announcement," http://blogs.adobe.com/conversations/2013/10/important-customer-security-announcement.html, October 2013, retrieved 2/10/2014.

[8] "ebay suffers massive security breach, all users must change their passwords," http://www.forbes.com/sites/gordonkelly/2014/05/21/ebay-suffers-massive-security-breach-all-users-must-their-change-passwords/, May 2014.

[9] S. Alexander, "Password protection for modern operating systems," *;login*, June 2004.

[10] L. C. Almeida, E. R. Andrade, P. S. Barreto, and M. A. Simplicio Jr, "Lyra: Password-based key derivation with tunable memory and processing costs," *Journal of Cryptographic Engineering*, vol. 4, no. 2, pp. 75–89, 2014.

[11] R. Biddle, S. Chiasson, and P. Van Oorschot, "Graphical passwords: Learning from the first twelve years," *ACM Computing Surveys (CSUR)*, vol. 44, no. 4, p. 19, 2012.

[12] A. Biryukov, D. Dinu, and D. Khovratovich, "Fast and tradeoff-resilient memory-hard functions for cryptocurrencies and password hashing," Cryptology ePrint Archive, Report 2015/430, 2015, http://eprint.iacr.org/.

[13] J. Blocki, M. Blum, and A. Datta, "Gotcha password hackers!" in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM, 2013, pp. 25–34.

[14] ——, "Naturally rehearsing passwords," in *Advances in Cryptology - ASIACRYPT 2013*, ser. Lecture Notes in Computer Science, K. Sako and P. Sarkar, Eds., vol. 8270. Springer Berlin Heidelberg, 2013, pp. 361–380. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-42045-0_19

[15] J. Blocki, N. Christin, A. Datta, A. D. Procaccia, and A. Sinha, "Audit games," in *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. AAAI Press, 2013, pp. 41–47.

[16] J. Blocki, A. Datta, and J. Bonneau, "Differentially private password frequency lists," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016*, 2016.

[17] J. Blocki, S. Komanduri, L. F. Cranor, and A. Datta, "Spaced repetition and mnemonics enable recall of multiple strong passwords," *CoRR*, vol. abs/1410.1490, 2014. [Online]. Available: http://arxiv.org/abs/1410.1490

[18] J. Blocki, S. Komanduri, A. Procaccia, and O. Sheffet, "Optimizing password composition policies," in *Proceedings of the fourteenth ACM conference on Electronic commerce*. ACM, 2013, pp. 105–122.

[19] J. Blocki and A. Sridhar, "Client-cash: Protecting master passwords against offline attacks," in *AsiaCCS (to appear)*. ACM, 2016.

[20] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 538–552.

[21] J. Bonneau and S. Preibusch, "The password thicket: technical and market failures in human authentication on the web," in *Proc. of WEIS*, vol. 2010, 2010.

[22] J. Bonneau and S. Schechter, ""toward reliable storage of 56-bit keys in human memory"," in *Proceedings of the 23rd USENIX Security Symposium*, August 2014.

[23] X. Boyen, "Halting password puzzles," in *Proc. Usenix Security*, 2007.

[24] S. Boztas, "Entropies, guessing, and cryptography," *Department of Mathematics, Royal Melbourne Institute of Technology, Tech. Rep*, vol. 6, 1999.

[25] J. G. Brainard, A. Juels, B. Kaliski, and M. Szydlo, "A new two-server approach for authentication with short secrets." in *USENIX Security*, vol. 3, 2003, pp. 201–214.

[26] S. Brostoff and M. Sasse, "Are Passfaces more usable than passwords: A field trial investigation," in *People and Computers XIV-Usability or Else: Proceedings of HCI*, 2000, pp. 405–424.

[27] J. Camenisch, A. Lysyanskaya, and G. Neven, "Practical yet universally composable two-server password-authenticated secret sharing," in *Proceedings of the 2012 ACM conference on Computer and Communications Security*. ACM, 2012, pp. 525–536.

[28] R. Canetti, S. Halevi, and M. Steiner, "Mitigating dictionary attacks on password-protected local storage," in *Advances in Cryptology-CRYPTO 2006*. Springer, 2006, pp. 160–179.

[29] V. Conitzer and T. Sandholm, "Computing the optimal strategy to commit to," in *Proceedings of the 7th ACM Conference on Electronic Commerce*. ACM, 2006, pp. 82–90.

[30] W. Daher and R. Canetti, "Posh: A generalized captcha with security applications," in *Proceedings of the 1st ACM workshop on Workshop on AISec*. ACM, 2008, pp. 1–10.

[31] S. Designer, "John the Ripper," http://www.openwall.com/john/, 1996-2010.

[32] K. Doel, "Scary logins: Worst passwords of 2012 and how to fix them," SplashData, 2012, retrieved 1/21/2013. [Online]. Available: http://www.prweb.com/releases/2012/10/prweb10046001.htm

[33] C. Dwork, A. Goldberg, and M. Naor, "On memory-bound functions for fighting spam," in *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2729. Springer, 2003, pp. 426–444. [Online]. Available: http://www.iacr.org/cryptodb/archive/2003/CRYPTO/1266/1266.pdf

[34] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of Cryptography*. Springer, 2006, pp. 265–284.

[35] C. Forler, S. Lucks, and J. Wenzel, "Catena: A memory-consuming password scrambler." *IACR Cryptology ePrint Archive*, vol. 2013, p. 525, 2013.

[36] M. Fossi, E. Johnson, D. Turner, T. Mack, J. Blackbird, D. McKinney, M. K. Low, T. Adams, M. P. Laucht, and J. Gough, "Symantec report on the underground economy," November 2008, retrieved 1/8/2013.

[37] D. Goodin, "Why passwords have never been weaker-and crackers have never been stronger," http://arstechnica.com/security/2012/08/passwords-under-assault/, August 2012.

[38] C. Herley and P. van Oorschot, "A research agenda acknowledging the persistence of passwords," *IEEE Symposium Security and Privacy*, vol. 10, no. 1, pp. 28–36, 2012.

[39] M. Hertzum, "Minimal-feedback hints for remembering passwords," *interactions*, vol. 13, pp. 38–40, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1125864.1125888

[40] M. Jain, E. Kardes, C. Kiekintveld, F. Ordónez, and M. Tambe, "Security games with arbitrary schedules: A branch and price approach." in *AAAI*, 2010.

[41] I. Jermyn, A. Mayer, F. Monrose, M. K. Reiter, and A. D. Rubin, "The design and analysis of graphical passwords," in *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*. Berkeley, CA, USA: USENIX Association, 1999, pp. 1–1. [Online]. Available: http://portal.acm.org/citation.cfm?id=1251421.1251422

[42] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2013.

[43] B. Kaliski, "Pkcs# 5: Password-based cryptography specification version 2.0," 2000.

[44] L. G. Khachiyan, "Polynomial algorithms in linear programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, no. 1, pp. 53–72, 1980.

[45] S. Komanduri, R. Shay, P. Kelley, M. Mazurek, L. Bauer, N. Christin, L. Cranor, and S. Egelman, "Of passwords and people: measuring the effect of password-composition policies," in *Proceedings of the 2011 annual conference on Human factors in computing systems*. ACM, 2011, pp. 2595–2604.

[46] U. Manber, "A simple scheme to make passwords based on one-way functions much harder to crack," *Computers & Security*, vol. 15, no. 2, pp. 171–176, 1996.

[47] J. Massey, "Guessing and entropy," in *Information Theory, 1994. Proceedings., 1994 IEEE International Symposium on*. IEEE, 1994, p. 204.

[48] R. Morris and K. Thompson, "Password security: A case history," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, 1979.

[49] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," *Advances in Cryptology-CRYPTO 2003*, pp. 617–630, 2003.

[50] C. Percival, "Stronger key derivation via sequential memory-hard functions," in *BSDCan 2009*, 2009.

[51] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," 2012.

[52] P. Pimsleur, "A memory schedule," *The Modern Language Journal*, vol. 51, no. 2, pp. pp. 73–75, 1967. [Online]. Available: http://www.jstor.org/stable/321812

[53] J. Pliam, "On the incomparability of entropy and marginal guesswork in brute-force attacks," *Progress in CryptologyINDOCRYPT 2000*, pp. 113–123, 2000.

[54] N. Provos and D. Mazieres, "Bcrypt algorithm."

[55] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions." in *Usenix security*. Baltimore, MD, USA, 2005, pp. 17–32.

[56] D. Seeley, "Password cracking: A game of wits," *Communications of the ACM*, vol. 32, no. 6, pp. 700–703, 1989.

[57] C. Shannon and W. Weaver, *The mathematical theory of communication*. Citeseer, 1959.

[58] R. Shay, P. Kelley, S. Komanduri, M. Mazurek, B. Ur, T. Vidas, L. Bauer, N. Christin, and L. Cranor, "Correct horse battery staple: Exploring the usability of system-assigned passphrases," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 2012, p. 7.

[59] Z. Yin, D. Korzhyk, C. Kiekintveld, V. Conitzer, and M. Tambe, "Stackelberg vs. nash in security games: Interchangeability, equivalence, and uniqueness," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2010, pp. 1139–1146.

[60] A. Zonenberg, "Distributed hash cracker: A cross-platform gpu-accelerated password recovery system," *Rensselaer Polytechnic Institute*, p. 27, 2009.

Missing Proofs

Reminder of Theorem 1. *We can find the solutions to Optimization Goal 2 in polynomial time in m, n and L, where L is the bit precision of our inputs.*

Proof of Theorem 1. (sketch) We first note that the convex feasible space from Optimization Goal 2 fits inside a ball of radius one. Thus, the Ellipsoid algorithm [44] will converge after making $poly(m)$ many queries to our separation oracle. By Theorem 2 the running time of the separation oracle is $O(mn \log mn)$. Thus, the total running time is polynomial in m and n. □

.1 Separation Oracle

The key idea behind Theorem 1 is to develop a polynomial time separation oracle. A separation oracle is an algorithm that takes as input a convex set $K \subseteq \mathbb{R}^m$ and a point $p \in \mathbb{R}^m$. The separation oracle outputs "Ok" if $p \in K$; otherwise it returns hyperplane separating x from K. In our context, the separation oracle takes as input a proposed solution $\mathcal{P}'_{Adv,B}, \tilde{p}'_1, \ldots, \tilde{p}'_m$ and outputs "Ok" if every constraint from Optimization Goal 2 is satisfied; otherwise the separation oracle finds a constraint that is not satisfied. If we can develop a polynomial time separation oracle for our linear program then we can use the ellipsoid algorithm to solve our linear program in polynomial time [44]. For our purposes, it is not necessary to understand how the ellipsoid algorithm works. Will we treat the ellipsoid algorithm as a blackbox that can solve a linear program in polynomial time given oracle access to a separation oracle.

We now present a separation oracle for Goal 2. Theorem 2 states that Algorithm 5 is a polynomial time separation oracle. We provide intuition for our separation oracle below. Theorem 1 follows immediately because we can run the ellipsoid algorithm [44] with our separation oracle to solve Goal 2 in polynomial time.

Theorem 2. *Algorithm 5 is valid separation oracle for Goal 2 and runs in time $O(mn \log mn)$.*

Algorithm 5 Separation Oracle. Output is an unsatisfied constraint C or "Ok" if every constraint is satisfied.

Input: $p_1, \ldots, p_n$, $\tilde{p}'_1, \ldots, \tilde{p}'_m$, B, C_{max}, k, α, and and $\mathcal{P}'_{Adv,B}$.

if $\sum_{i=1}^{m} \tilde{p}'_i \neq 1$ **then**
  return $\sum_{i=1}^{m} \tilde{p}_i = 1$.
end if
if $(1-\alpha)m \cdot k + k \cdot \alpha \sum_{i=1}^{m} i \cdot \tilde{p}'_i > C_{max}$ **then**
  return $(1-\alpha)m \cdot k + k \cdot \alpha \sum_{i=1}^{m} i \cdot \tilde{p}'_i \leq C_{max}$
end if
for i=1,...,m **do**
  if $\tilde{p}'_i < 0$ **then**
    return $\tilde{p}_i \geq 0$.
  end if
  if $i < m$ and $\tilde{p}'_{i+1} > \tilde{p}'_i$ **then**
    return $\tilde{p}_{i+1} \leq \tilde{p}_i$.
  end if
end for
if $\mathcal{P}'_{Adv,B} > 1$ **then**
  return $\mathcal{P}'_{Adv,B} \leq 1$
end if
if $\mathcal{P}'_{Adv,B} < 0$ **then**
  return $\mathcal{P}'_{Adv,B} \geq 0$
end if
for i = 1,..., n **do**
  for j = 1,..., m **do**
    $p'_{i,j} \leftarrow p_i \tilde{p}'_j$.
  end for
end for
$TUPLES \leftarrow \{(i,j) \,|1 \leq i \leq n \wedge 1 \leq j \leq m\}$.
Define ordering $\succ$ over $TUPLES$: $(i_1, j_1) \succ (i_2, j_2)$ if any of the following conditions hold (1) $p_{i_1,j_1} > p_{i_2,j_2}$, or (2) $p_{i_1,j_1} = p_{i_2,j_2}$ and $i_1 < i_2$ or (3)$p_{i_1,j_1} = p_{i_2,j_2}$ and $i_1 = i_2$ and $j_1 < j_2$.
$SORTED-TUPLES \leftarrow$ **SORT** $(TUPLES, \succ)$. {Let $T_k \doteq SORTED-TUPLES[k]$. } {$T_k$ is the k'th biggest element according to $\succ$}
$S \leftarrow \{T_1, \ldots, T_B\}$.
for i=1,...,n **do**
  $b'_i \leftarrow \max\{j \in \mathbb{Z} \,|j = 0 \vee (i,j) \in S\}$
end for
if $\mathcal{P}'_{Adv,B} < \sum_{i=1}^{n} p_i \sum_{j=1}^{b'_i} \tilde{p}'_j$ **then**
  return $\mathcal{P}_{Adv,B} \geq \sum_{i=1}^{n} p_i \sum_{j=1}^{b'_i} \tilde{p}_j$
else
  return "Ok"
end if

Intuitively, the idea behind the separation oracle is quite simple. Suppose that we want to verify that the variable assignment $\tilde{p}'_1, \ldots, \tilde{p}'_m, \mathcal{P}'_{Adv,B}$ is feasible. The first few steps of our separation oracle verify that constraints (1)–(4) from Goal 2 are satisfied by the assignment $\tilde{p}'_1, \ldots, \tilde{p}'_m$. These straightforward checks simply verify that the proposed CASH distribution $\tilde{p}'_1, \ldots, \tilde{p}'_m$ is valid and that the server's amortized costs are less than $C_{SRV,\alpha}$.

The next step, verifying that all type (5) constraints are satisfied, is a bit more challenging because there are exponentially many constraints. Recall that these constraints intuitively ensure that $\mathcal{P}'_{Adv,B}$ is indeed an upper bound on the success rate of the optimal adversary given CASH distribution $\tilde{p}'_1, \ldots, \tilde{p}'_m$. While

we don't have time to check every feasible budget allocation $\vec{b} \in \mathcal{F}_B$ for the adversary, it suffices to find the adversary's optimal budget allocation $\vec{b}'$ and verify that $\mathcal{P}'_{Adv,B}$ is an upper bound on the adversary's success rate given allocation $\vec{b}'$.

The adversary gets $\lfloor B/k \rfloor$ total guesses of the form (pw_i, j) for each user u. The probability that the guess (pw_i, j) is correct is simply $p'_{i,j} \doteq p_i \cdot \tilde{p}'_j$ — the guess is correct if and only if u selected password $pwd_u = pwd_i$ and we selected CASH running time parameter $t_u = j$. The adversary's optimal strategy is simple: try the $\lfloor B/k \rfloor$ most likely guesses. Thus, we can quickly find the adversary's optimal budget allocation $\vec{b}'$ by computing $p_{i,j}$ for each pair (pw_i, j) and sorting these values. This takes time $O(nm \log nm)$.

Reminder of Theorem 2. *Algorithm 5 is valid separation oracle for Goal 2 and runs in time $O(mn \log mn)$.*

Proof of Theorem 2. (Sketch) The most expensive step in our algorithm is sorting the $p'_{i,j}$ values. There are mn such values so the algorithm takes $O(mn \log mn)$ steps. We now argue that our separation oracle has correct behavior.

Suppose first that there is a constraint C from Optimization Goal 2 that is not satisfied by $\tilde{p}'_1, \ldots, \tilde{p}'_m, \mathcal{P}'_{ADV,B}$. It is easy to verify that our separation oracle will catch violations of constraints (1)–(4) so we can assume that C be a violated type (5) constraint $\mathcal{P}'_{ADV,B} < \sum_{i=1}^{n} p_i \sum_{j=1}^{b_i^C} \tilde{p}_j$ where $\left(b_1^C, \ldots, b_n^C\right) \in \mathcal{F}_B$. Let $b'_1, \ldots, b'_n$ denote the budget obtained by sorting the $p_{i,j}$ values and then greedily selecting a set S' of the largest values until the budget expires — we define b'_i to be the number of values of the form $p_{i,j}$ that are selected and $S' = \{(i,j) \mid i \leq n \wedge j \leq b'_i\}$ It suffices to argue that

$$\sum_{i=1}^{n} p_i \sum_{j=1}^{b'_i} \tilde{p}_j \geq \sum_{i=1}^{n} p_i \sum_{j=1}^{b_i^C} \tilde{p}_j$$

because in this case our algorithm will return the violated constraint

$$\mathcal{P}'_{ADV,B} \geq \sum_{i=1}^{n} p_i \sum_{j=1}^{b'_i} \tilde{p}_j \ .$$

Let $S^C = \{(i,j) \mid i \leq n \wedge j \leq b_i^C\}$. We first observe that

$$\sum_{(i,j) \in S'} p'_{i,j} \geq \sum_{(i,j) \in S^C} p'_{i,j} \ ,$$

by construction of S'. Thus,

$$\begin{aligned} \sum_{i=1}^{n} p_i \sum_{j=1}^{b'_i} \tilde{p}'_j &= \sum_{(i,j) \in S'} p'_{i,j} \\ &\geq \sum_{(i,j) \in S^C} \\ &= \sum_{i=1}^{n} p_i \sum_{j=1}^{b_i^C} \tilde{p}'_j \end{aligned}$$

Finally, when the solution $\tilde{p}'_1, \ldots, \tilde{p}'_m, \mathcal{P}'_{ADV,B}$ does satisfy all constraints from Optimization Goal 2 our algorithm will not find a constraint $b'_1, \ldots, b'_n$ such that

$$\mathcal{P}'_{ADV,B} < \sum_{i=1}^{n} p_i \sum_{j=1}^{b'_i} \tilde{p}_j \ .$$

In this case our algorithm will return "Ok" — the desired outcome. □

Algorithm 6 InitialConstraints (C_{max}, α, k)

Input: C_{max}, α, k

$C \leftarrow C \bigcup \{\sum_{i=1}^{m} \tilde{p}_i = 1\}$.
$C \leftarrow C \bigcup \{1 \geq \tilde{p}_m \geq 0\}$.
$C \leftarrow C \bigcup \{(1-\alpha)m \cdot k + \alpha \cdot k \sum_{i=1}^{m} i \cdot \tilde{p}_i \leq C_{SRV,\alpha}\}$.
for i=1,...,m-1 **do**
  $C \leftarrow C \bigcup \{1 \geq \tilde{p}_i \geq 0\}$.
  $C \leftarrow C \bigcup \{\tilde{p}_i \geq \tilde{p}_{i+1}\}$.
end for
$C \leftarrow C \bigcup \{1 \geq \mathcal{P}_{Adv,B} \geq 0\}$.

Algorithm 7 RationalAdvSuccess $(p, n, \hat{v}, \tilde{p}, k)$

Input: $p_1, \ldots, p_{n'}$, $n_1, \ldots, n_{n'}$, $\hat{v}$, $\tilde{p}$

$curSuccess \leftarrow 0$
$curThreshold \leftarrow 0$
$curUtility \leftarrow 0$
$bestUtilityFound \leftarrow 0$
$bestUtilitySuccess \leftarrow 0$
$T \leftarrow \emptyset$
for $i = 1, \ldots, n'$ **do**
  for $j = 1, \ldots, m$ **do**
    $T.Add\,(p_i \cdot \tilde{p}_j, n_i)$
  end for
end for
Sort (T). {Use first component $p_i \cdot \tilde{p}_j$ for}
{ comparison (greatest to least)}
for $t \in T$ **do**
  $(\pi, count) \leftarrow t$
  $curThreshold \leftarrow curThreshold + count$
  $curSuccess \leftarrow \pi \cdot count$
  $\Delta benefit \leftarrow \hat{v} \cdot \pi \cdot count$
  $\Delta cost \leftarrow k * \left(count * (1 - curSuccess) + \frac{\pi \cdot count^2 + \pi \cdot count}{2}\right)$
  $curUtility \leftarrow curUtility + \Delta benefit - \Delta cost$
  if $curUtility > bestUtilityFound$ **then**
    $bestUtilityFound \leftarrow curUtility$
    $bestUtilitySuccess \leftarrow curSuccess$
  end if
end for
return $bestUtilitySuccess$

While we do not have a polynomial time algorithm to compute the Stackelberg equilibrium of our game, it is always easy for the adversary to compute his best response.

Theorem 3. *Let $p = p_1 \geq \ldots \geq p_{n'}$ and $n_1, \ldots, n_{n'}$ define a probability distribution over passwords in which there are n_i passwords that each are chosen with probability p_i and let $\tilde{p}_1 \geq \ldots \geq \tilde{p}_m$ denote any CASH distribution. Then for any value $\hat{v}$ and any hash cost parameter k we can computed the adversary's optimal strategy in time $O(mn' \log mn')$.*

Proof. (sketch) Algorithm 7 computes the adversay's optimal strategy. The most expensive step is the sorting the mn tuples, which takes time $O(mn' \log mn')$. Thus, Algorithm 7 runs in time in time $O(mn' \log mn')$. Algorithm 7 iterates through the different possible thresholds that a rational adversary might select. The variable *curUtility* keeps track of the utility at each threshold allowing us to remember which threshold was optimal. Intuitively, Algorithm 7 will find the best strategy if and only if *curUtility* is always a correct estimate of the adversary's utility. Clearly, this is true initially (the utility of selecting $B^* = 0$ is 0). Thus, by induction, it suffices to show that the formulas used to compute marginal cost and marginal benefit are correct. If the adversary adds all of the tuples (pwd, t) corresponding to $(\pi, count)$ to the set of tuples to guess then the adversary is increasing the odds that he cracks the password by $\pi \cdot count$ because he is adding *count* tuples to his set of guesses and each tuple is correct with probability π. Thus, his marginal benefit is $\hat{v} \cdot \pi \cdot count$. To analyze marginal cost we consider three cases: 1) The correct tuple $(pwd*, t*)$ was already in the adversary's set of tuples to guess. In this case we don't increase the adversary's guessing costs because he will always quit before he guesses one of the new tuples we added. 2) The correct tuple $(pwd*, t*)$ is not already in the adversary's set of tuples and it is not in the new set of tuples we add. Thus, we increase the adversary's guessing costs by $k * count$. 3) The correct tuple $(pwd*, t*)$ is in the new set of tuples we add. In this case we increase the adversary's guessing costs by $k * \left(\frac{count+1}{2}\right)$ in expectation. The probability that we are in case 2 is $(1 - curSuccess)$ and the probability that we are in case 3 is $\pi \cdot count$. Thus,

$$\Delta cost \leftarrow k * \left(count * (1 - curSuccess) + \frac{\pi \cdot count^2 + \pi \cdot count}{2} \right) .$$

□

Extra Plots

Figures 6 and 5 explore what happens when the defender uses the wrong empirical password distribution when searching for a good CASH distribution $\tilde{p}$ (e.g., if the defender optimizes $\tilde{p}$ under the assumption that the empirical password distribution is given by the Yahoo! dataset when the actual distribution is given by the RockYou dataset). Once again non-uniform CASH and CASH both significantly outperform deterministic key-stretching, an non-uniform CASH outperforms uniform CASH (slightly) over most of the curve[20]. Interestingly, in one part of the curve in Figure 6 the adversary's success rate actually drops as v increases. This would be impossible if the defender was using the correct empirical password distribution. In this case the adversary's success rate drops when v increases because the defender switches to a better CASH distribution $\tilde{p}$ that happens to perform better under the real distribution.

Figure 7 plots the fraction of cracked passwords against a value v adversary when the defender selects $\tilde{p}$ and k under the assumption that the adversary's value is $\hat{v} = 2.9 \times C_{max} \times 10^7$ (using the empirical password distribution from the Yahoo dataset and setting $\alpha = 0.95$). Figure 8 plots the corresponding cumulative cost distribution for the authentication server induced by $\tilde{p}$, k and α. For comparison, we also include the cumulative cost distributions for uniform CASH and deterministic key-stretching under the same maximum cost parameter C_{max}.

[20]The exception is Figure 6 contains a region where uniform-CASH actually outperforms non-uniform CASH (yielding 15% reduction in cracked passwords in comparison to non-uniform CASH).

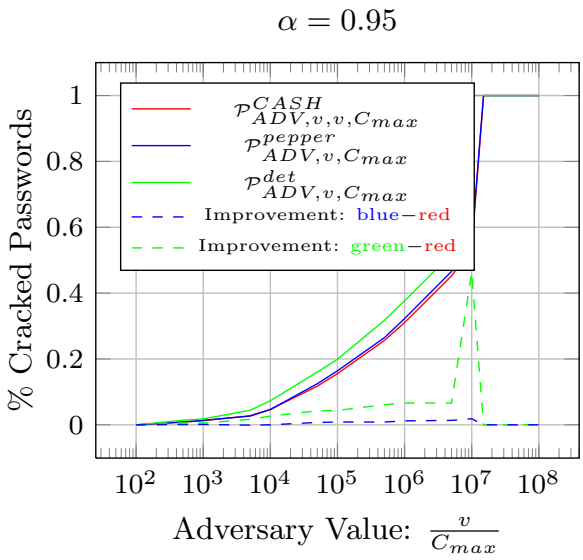


Figure 5: RockYou Results (Optimized for Yahoo): $\alpha = 0.95$.

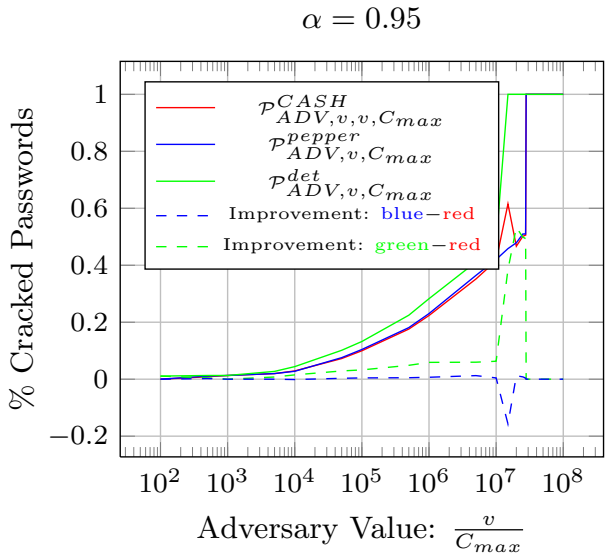


Figure 6: Yahoo Results (Optimized for RockYou): $\alpha = 0.95$.

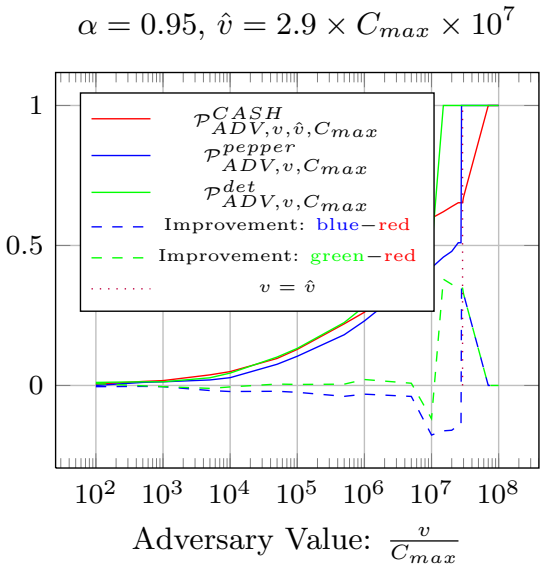


Figure 7: Yahoo: $\hat{v} \neq v$.

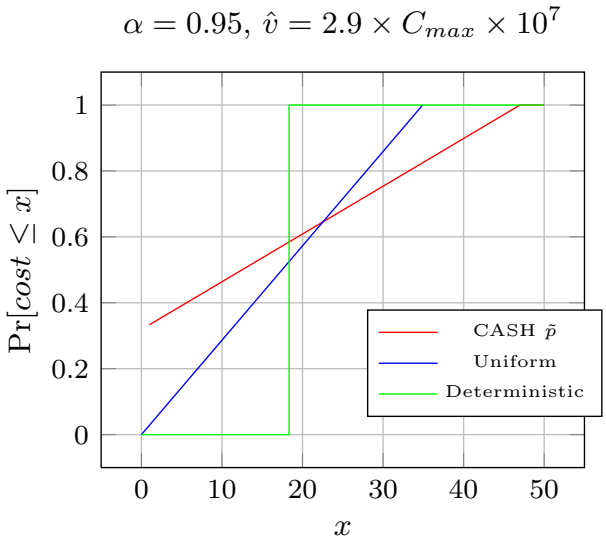


Figure 8: Yahoo: CASH Cumulative Probability Distribution.